PROCEEDINGS OF

OSPERT 2014

the 10th Annual Workshop on
Operating Systems Platforms for
Embedded Real-Time Applications

July 8th, 2014 in Madrid, Spain

in conjunction with

ECRTS

the 26th Euromicro Conference on Real-Time Systems
July 9–11, 2014, Madrid, Spain

Editors:
Björn B. Brandenburg
Shinpei Kato

Contents

Message from the Chairs **5**

Program Committee **5**

Keynote Talk **7**

Session 1: RTOS Design and Implementation I **9**

ARM-based SoC with Loosely Coupled Type Hardware RTOS for Industrial Network Systems
Naotaka Maruyama, Takuya Ishikawa, Shinya Honda, Hiroaki Takada, and Katsunobu Suzuki 9

Distributed Real-Time Fault Tolerance on a Virtualized Multi-Core System
Eric Missimer, Richard West, and Ye Li 17

Fast User Space Priority Switching
Alexander Zuepke, Marc Bommert, and Robert Kaiser 23

Session 2: Mixed-Criticality Systems **31**

Implications of Multi-Core Processors on Safety-Critical Operating System Architectures
Stefan Burger, Kevin Müller, Oliver Hanka, Michael Paulitsch, Andrea Bastoni, Henrik Theiling, and Matthias Heinisch 31

Towards Hard Real-Time Control and Infotainment Applications in Automotive Platforms
Mian M. Hamayun, Alexander Spyridakis, and Daniel S. Raho 39

Mixed-Criticality on Multicore (MC^2): A Status Report
Namhoon Kim, Jeremy P. Erickson, and James H. Anderson 45

Session 3: RTOS Design and Implementation II **51**

A Platform for LEGO Mindstorms EV3 Based on an RTOS with MMU Support
Yixiao Li, Takuya Ishikawa, Yutaka Matsubara, and Hiroaki Takada 51

Usable RTOS-APIs?
Tobias Klaus, Florian Franzmann, Tobias Engelhard, Fabian Scheler, and Wolfgang Schröder-Preikschat 61

Program **68**

Message from the Chairs

Welcome to Madrid, Spain, and to OSPERT'14, the 10th annual workshop on Operating Systems Platforms for Embedded Real-Time Applications. It has been a pleasure and honor to serve as the chairs for OSPERT's 10-year anniversary, and we invite you to celebrate with us the first decade of this unique venue for exchanging ideas about systems issues related to real-time and embedded systems. OSPERT's key strengths have always been the participants, representing a healthy mix of academics and industry experts, and the many interesting discussions facilitated by OSPERT's interactive format, which are two traditions that we aim to continue.

Paolo Gai will open the workshop with his keynote highlighting the challenges and opportunities in developing, establishing, and maintaining commercially viable open-source RTOS solutions for automotive systems. As founder and CEO of Evidence Srl, a successful spinoff of the well-known ReTiS Lab at Scuola Superiore Sant'Anna in Pisa, Paolo is in a unique position to discuss this topic and we are delighted that he volunteered to share his experience and perspective.

OSPERT this year accepted eight of nine peer-reviewed papers, which cluster around two central topics. First, OSPERT's core focus area—RTOS design and implementation—is well represented, with contributions ranging from a hardware-based RTOS, over fault tolerance in a multi-kernel OS, to reflections on API design, efficient kernel interfaces, and a report on an educational RTOS for LEGO Mindstorm devices. Second, reflecting the growing importance of efficiently supporting safety-critical workloads and workloads of mixed criticality on shared hardware platforms, this year's program features two timely reports on system design for safety-critical workloads in the avionics and automotive domains. The program is completed by an invited paper reporting on the progress of MC^2, an open-source platform for applied systems research on hosting mixed-criticality workloads on multicore platforms.

OSPERT 2014 would not have been possible without the support of many people. The first thanks are due to Gerhard Fohler and the ECRTS steering committee for entrusting us with organizing OSPERT'14, and for their continued support of the workshop. We would also like to thank the program committee for volunteering their time and effort to provide useful feedback to the authors, and of course all the authors for their contributions and hard work.

On the occasion of OSPERT's 10-year anniversary, we extend our special thanks to the chairs of prior editions of the workshop who shaped OSPERT and let it grow into the successful event that it is today.

Last, but not least, we thank you, the audience, for actively contributing—through your stimulating questions and lively interest—to define and improve OSPERT. We hope you will enjoy this day.

The Workshop Chairs,

Björn B. Brandenburg
Max Planck Institute for Software Systems
Kaiserslautern, Germany

Shinpei Kato
Nagoya University
Nagoya, Japan

Program Committee

Keynote Talk

Open-source and Real-time in Automotive Systems: (not only) Linux, (not only) AUTOSAR

Paolo Gai
Evidence Srl

The talk will consider the current status of open-source and real-time in automotive systems, starting from the history of the open-source real-time OS ERIKA Enterprise. The current trends will also be considered including the usage of open-source Linux systems and the integration and safety qualification issues on modern automotive multicores.

Dr. Paolo Gai, CEO of Evidence Srl, graduated (cum laude) in Computer Engineering at University of Pisa in 2000 with a graduation thesis developed at the ReTiS Laboratory of the Scuola Superiore SantAnna on the development of the modular real-time kernel SHaRK. He obtained the PhD from Scuola Superiore Sant'Anna in 2004. Since 2000, he founded the ERIKA Enterprise project, an open-source RTOS which recently reached the OSEK/VDX certification, and which is currently used by various industries and universities. Since 2002 he is CEO and founder of Evidence Srl, a SME working on operating systems and code generation for Linux- and ERIKA-based industrial products in the automotive and white goods market. His research interests include development of hard real-time architectures for embedded control systems, multi-processor systems, object-oriented programming, real-time operating systems, scheduling algorithms and multimedia applications.

ARM-based SoC with Loosely coupled type hardware RTOS for industrial network systems

Naotaka Maruyama*†, Takuya Ishikawa†, Shinya Honda†, Hiroaki Takada† and Katsunobu Suzuki‡
*KERNELON Silicon Inc., 520-6, Fujisawa, Fujisawa-shi, 251-0052, Japan
Email: maruyama_na@kernelon.com
†Graduate School of Information Science, Nagoya University, Furo-cho, Chikusa-ku, Nagoya, 464-8603, Japan
Email: {t_ishikawa, honda, hiro}@ertl.jp
‡2nd Solution Business Unit, Renesas Electronics Corp., 1753, Simonumabe, Nakahara-ku, Kawasaki-shi, 211-8668, Japan
Email: katsunobu.suzuki.fn@renesas.com

***Abstract*—Many different types of high-speed networks are employed in industrial systems, which affect real-time processing, such as motor control of industrial controllers, because of the increased CPU load due to network protocol processing. In this study, we propose a system-on-a-chip (SoC) architecture for industrial controllers to reduce the network protocol processing load. The proposed architecture adopts a RTOS in hardware in order to accelerate RTOS execution because the RTOS is frequently called in protocol processing and commonly used in many protocols. Existing hardware RTOSs require a purpose-built core which has a special interface to connect with the hardware RTOS in a tightly coupled manner. However, ARM processors are required in many industrial systems. We manufactured a SoC for Industrial controllers using the proposed architecture and it is the world's first SoC with hardware RTOS, and it is commercially available. The results of our experimental evaluations showed that the API execution time of the proposed architecture was 1.4–2.9 times faster and the UDP/IP throughput of the proposed architecture was 1.67 times faster compared with that when using a conventional software RTOS.**

I. Introduction

In factories, low speed networks such as controller area networks (CANs) have been used as industrial networks. However, high-speed networks such as 10/100/1000 Mbps Ethernet have been employed recently. Industrial controllers (IndCntlrs), which control industrial devices such as motors, require real-time processing. However, the deployment of high-speed network systems affects the real-time processing of IndCntlrs. High-speed network protocol processing occupies a large amount of CPU time and generates frequent interrupts, thus it is necessary to decrease the network protocol processing load of IndCntlrs.

In contrast to office networks, industrial networks are required to meet real-time constraints. This requires periodic and deterministic data transfer to ensure synchronization among the IndCntlrs connected by the network. An Ethernet is adopted as a physical layer in industrial network protocol stacks but many protocols, such as PROcess FIeld NETwork (PROFINET), Ethernet industrial protocol (EtherNet/IP), ModbusTCP, and Ethernet for control automation technology (EtherCAT), have been proposed for use as the upper layer of industrial network protocol stacks to satisfy real-time constraints[14]. Therefore, multi-protocol support is required for IndCntlrs. Furthermore, improved real-time processing is also required to satisfy the real time constraints.

The adoption of ARM cores is required for IndCntlr because users want to utilize the properties of software that have already been developed and they need to retain the same integrated development environments. Furthermore, ARM cores provide scalability because they have many lineups and they are also reliable due to the fact they have been implemented in a huge number of embedded systems throughout the world. For these reasons, major semiconductor manufacturers have moved their products from proprietary cores to ARM cores.

Low power consumption and low costs are important because industrial systems comprise a large number of IndCntlrs, which are connected via a network.

As mentioned above, the requirements for industrial network systems are: (1) decreasing the load of network protocol processing, (2) improved real-time processing, (3) multi-protocol support, (4) adoption of ARM cores, and (5) low costs and low power consumption.

One approach that satisfies requirements (1) and (2) is to adopt a network protocol offload engine. However, to satisfy requirement (3), the engines have to be designed and implemented for all protocol types. Furthermore, this approach lacks flexibility, needs a greater silicon area in the SoC, and is more expensive; thus, it is impractical. Another approach is to increase the clock rate of the core. However, requirement (5) is not satisfied because the system has high costs and high power consumption.

The purpose of the present study is to propose an architecture for a SoC for IndCntlrs that satisfy the requirements mentioned above and to manufacture the SoC. We satisfied the requirements by implementing a RTOS in hardware. The RTOS APIs are invoked frequently during protocol processing in any type of protocol. Thus, a large amount of CPU time is consumed by RTOS execution during the protocol processing.

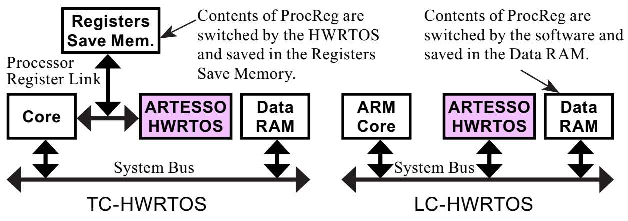

Registers Save Mem.
Contents of ProcReg are switched by the HWRTOS and saved in the Registers Save Memory.
Contents of ProcReg are switched by the software and saved in the Data RAM.
Processor Register Link
Core
ARTESSO HWRTOS
Data RAM
ARM Core
ARTESSO HWRTOS
Data RAM
System Bus
System Bus
TC-HWRTOS
LC-HWRTOS

Fig. 1. **Coupling types for the HWRTOS**

Therefore, if a RTOS is implemented in hardware and the CPU time occupation by RTOS is reduced, the protocol processing load can be reduced. Thus, requirement (1) is satisfied. Requirement (2) is also satisfied because reducing the RTOS overheads also decreases the interrupt response time. This method also satisfies requirement (3) because the hardware RTOS is commonly used in many type of protocols, thus it decreases the processing load for many type of protocols. Furthermore, the method satisfies requirement (5) because it does not require an increase in the core performance. In this study, we refer to ”RTOS implemented in hardware” as HWRTOS and ”RTOS implemented in software” as SWRTOS.

As shown in Fig.1, there are two methods for connecting a core with a HWRTOS. One is a tightly coupled type of HWRTOS (TC-HWRTOS) and the other is a loosely coupled type of HWRTOS (LC-HWRTOS). Requirement (4) is satisfied with LC-HWRTOS, as mentioned in Section II, thus we adopted the LC-HWRTOS.

The main contributions of this study are as follows.

1. Proposal of a LC-HWRTOS architecture for IndCntlr.

2. Proposal of improvement of performance by parallel execution of a core and a HWRTOS.

3. Design and manufacture of a SoC for IndCntlrs.

4. Evaluation of the RTOS performance and network throughputs of both LC-HWRTOS and SWRTOS on the SoC.

The remainder of this paper is organized as follows. The HWRTOS architecture for IndCntlrs is described in Section II. Section III provides details of the LC-HWRTOS. Section IV explains the architecture of the SoC and the performance evaluations are presented in Section V. Related work is presented in Section VI and Section VII concludes this study. The Appendix describes our previous development system, original ARTESSO system[12]. The HWRTOS that we developed is referred to as ARTESSO HWRTOS.

II. HWRTOS ARCHITECTURE FOR INDCNTLR

This section describes why the LC-HWRTOS is adopted to satisfy requirement (4).

The following are the combinations of four cores and RTOS types for the IndCntlr SoC.

(A) HWRTOS + purpose-built core	(TC-HWRTOS)
(B) HWRTOS + modified ARM core	(TC-HWRTOS)
(C) HWRTOS + ARM core	(LC-HWRTOS)
(D) SWRTOS + ARM core	(SWRTOS)

The following explains the TC-HWRTOS and LC-HWRTOS as shown in Fig.1. In TC-HWRTOS, the core has the Processor Register Link, which is a special interface

TABLE I
Different combinations of RTOSs and cores

Combination Type	(A)	(B)	(C)	(D)
RTOS	TC-HWRTOS		LC-HWRTOS	SWRTOS
Core	Purpose-build	ARM (modified)	ARM	
API execution time	Low		Middle	High
Tick management offloading	Available			Un-available
IIA offloading	Available			Un-available
Core Reliability	Middle		High	
Core Scalability	No	Low	Yes	
Core verification cost	High	Middle	Low	
Standard Tools	Unusable	Usable		
LSI development cost	Middle	High	Middle	Low

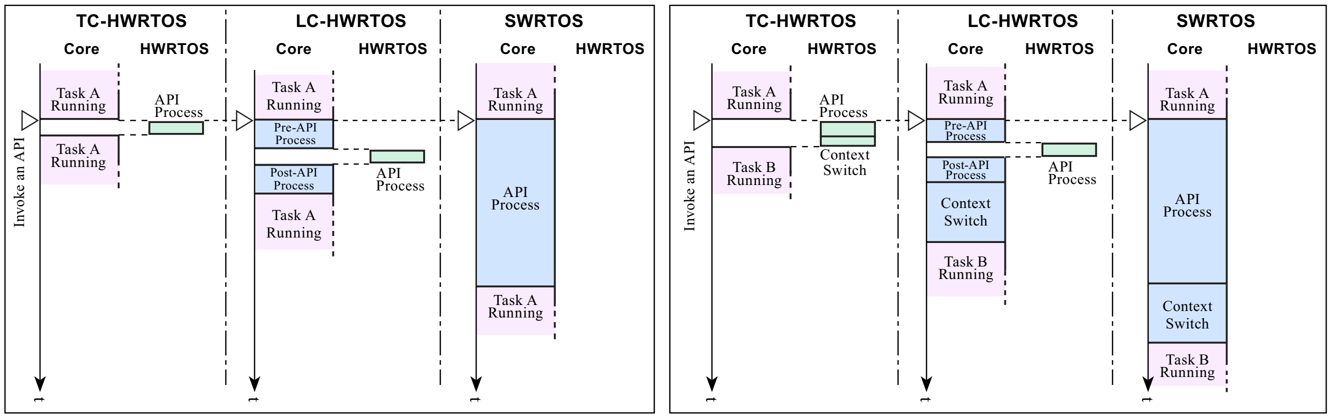
for calling an API and for switching the contents between the Register Save Memory and the internal registers of the core. The internal registers comprise a program counter, stack pointer, flag register, and general-purpose registers. In the present study, the core internal registers are referred to as ProcReg. The contents of the ProcReg are kept in the Register Save Memory on a task-by-task basis. Thus, all API functions, including context switching, are implemented in hardware. Therefore, in TC-HWRTOS, the execution time is quite fast, although a purpose-built core has to be used. As mentioned in the Appendix, the original ARTESSO system is configured as a TC-HWRTOS and it belongs to type (A). (B) is configured as a TC-HWRTOS with an ARM core, thus the ARM core needs to be modified to provide a special interface with ARTESSO HWRTOS. However, the scalability and reliability of the core are lower than formal ARM products since it is modified.

In LC-HWRTOS, the HWRTOS is implemented on the system bus of the core, thus the core can invoke an API through the system bus and the core does not need to be modified with a special interface. Therefore, the API execution is quite fast, although context switching is executed by software in the same manner as SWRTOS.

Table I shows the advantages and disadvantages of the each combination. (C) is configured as a LC-HWRTOS with an ARM core. (D) is a conventional SWRTOS system.

The following describes comparison of the API execution sequence for each RTOS type. Fig.2 (a) shows the sequence without context switching. In TC-HWRTOS, after invoking an API, the HWRTOS executes the API function and the task restarts after completion. In contrast to the TC-HWRTOS, the LC-HWRTOS requires a pre-procedure to call APIs and a post-procedure to obtain the result value, because arguments and a return value of the API are handed over using registers in LC-HWRTOS, which are deployed between the core and the ARTESSO HWRTOS, as mentioned in Section III, whereas they can be handed over using ProcReg in TC-HWRTOS. Fig.2 (b) shows the API function sequence with context switching. The LC-HWRTOS requires a context-switching process based on software in addition to the pre- and post-procedures. The software used to call the RTOS, such as pre- and post-procedures, is called a RTOS driver.

To satisfy requirement (4), (B) or (C) must be selected. The execution of APIs by (B) is faster than that by (C). However, the scalability and the reliability are lower than formal ARM,



(a) API without context switch (b) API with context switch

Fig. 2. **API function sequence**

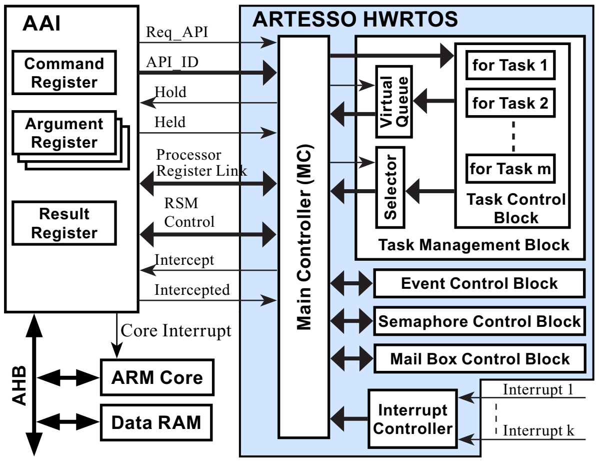


Fig. 3. **LC-HWRTOS for the IndCntlr SoC**

and development cost is high in (B). The core availability and real-time response in (C) are improved compared with (D) because all of the RTOS functions are executed by hardware, except for context switching. Based on these considerations, we selected (C), the LC-HWRTOS, as the RTOS for the IndCntlr SoC.

III. LC-HWRTOS FOR THE INDCNTLR SOC

A. Architecture of the LC-HWRTOS

Fig.3 shows the architecture of the IndCntlr SoC based on the LC-HWRTOS method. The original ARTESSO system was designed based on the TC-HWRTOS, and the LC-HWRTOS of this study was modified the original ARTESSO system. The modifications are as follows. A new module, ARTESSO AHB Interface (AAI) is implemented, which permits the ARM core to access the ARTESSO HWRTOS through the AHB. AHB stands for advanced high-performance bus and is ARM system bus. The AAI includes the ”Command Register,” ”Argument Registers,” and ”Result Register.” The core accesses these registers through the AHB to invoke an API and to obtain the return value of the API. The Argument Registers and Result Register are also accessed from the Main Controller (MC). The MC is modified to connect with the AAI and to implement new functions, as described in subsection C.

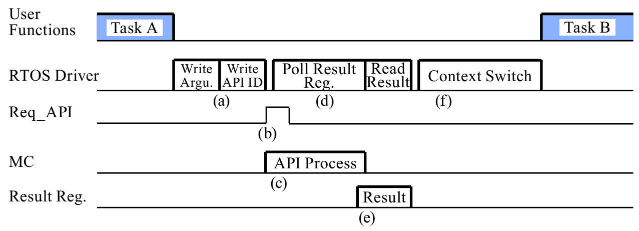


Fig. 4. **API call sequence in LC-HWRTOS**

The following modules are same as those in the original ARTESSO system. The Task Control Block maintains the management data related to all of the tasks, such as the ”current state” and ”task priority.” The Virtual Queue implements a great deal of queues in small hardware and provides fast queue operations. The Event Control Block, the Semaphore Control Block, and the Mail Box Control Block maintain and manage the information required by event functions, semaphore functions, and mailbox functions, respectively.

B. Procedure for Calling an API

Fig.4 shows the API call sequence. The core writes the argument into the Argument Registers and writes an API identifier in the Command Register, (a). Next, the AAI sends a Req_API signal to the MC with an API_ID signal which indicates the API identifier, (b). When the signals are detected, the MC begins the execution of the API, (c). After the MC completes the execution of the API, the MC writes the return value into the Result Register, (e). The core polls the Result Register after the API call and it reads the value until a valid return value is written into the Result Register by the MC, (d).

If the API execution requires context switching, information that indicates the context switching request and the dispatched task identifier are written in the Result Register with the return value. The core then executes context switching using software, (f). Specifically, the contents of the current task in the ProcReg in the core are saved to the Data RAM and the contents of the next task are loaded into the ProcReg.

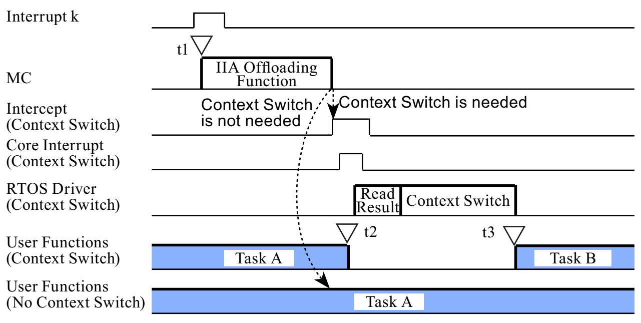

Interrupt k
t1
MC
IIA Offloading Function
Intercept (Context Switch)
Context Switch is not needed
Context Switch is needed
Core Interrupt (Context Switch)
RTOS Driver (Context Switch)
Read Result
Context Switch
t2
t3
User Functions (Context Switch)
Task A
Task B
User Functions (No Context Switch)
Task A

Fig. 5. **IIA sequence in POHC**

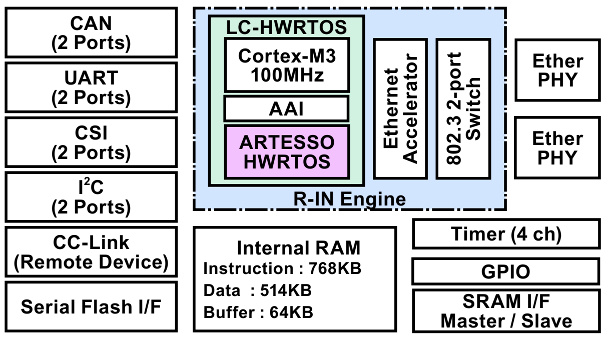


Fig. 6. **Overall configuration of R-IN32M3 SoC**

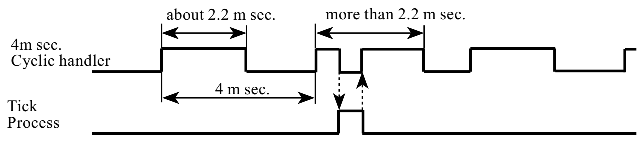


Fig. 7. **Test of the effect of the time tick**

C. *Parallel execution of core and ARTESSO HWRTOS*

As mentioned in Appendix, the original ARTESSO HWRTOS implements not only API functions but also other RTOS functions such as interrupt invoked API (IIA) and tick management to offload their works from the core. When an interrupt occurs, the IIA function invokes a selected API, which is determined by the interrupt cause. After the IIA function the MC executes task switch if needed. Fig.11 in the Appendix shows the IIA offloading in the TC-HWRTOS and Fig.10 (1) shows that the SWRTOS executes the same procedure as IIA offloading. In the tick management function, the ARTESSO HWRTOS implements hardware timeout-timers for each task and if the MC detects a timer expiration, the MC moves the task from the wait queue to the ready queue, and then the MC executes task switch if needed.

In the SWRTOS, the user function is suspended while executing the RTOS function. Since the RTOS functions and user functions are both executed on a core and the user function has to wait for return value of called API. In the original ARTESSO HWRTOS, the user function was suspended during the MC working in the same manner as the SWRTOS. This method is called serial operation of a HWRTOS and a core (SOHC). However the IIA and the tick management functions are invoked by interrupts therefore they can be started without invocation by user functions executing on the core, thus the IIA and the tick management can execute in parallel with user functions in the HWRTOS. Proposed architecture allows parallel operation by modifying the MC. This method is called parallel operation of a HWRTOS and a core (POHC). If the RTOS decides that a task switching is not needed at the end of the IIA or tick management function, the currently executing user function continues to run, as shown in Fig.5. The POHC improves the system performance, especially in systems where interrupts are generated frequently such as network protocol processing.

IV. SoC implementation: R-IN32M3

This section provides a summary of the R-IN32M3 SoC[15], which we developed as an IndCntlr to satisfy the requirements specified in Section I. Fig.6 shows the overall configuration of R-IN32M3. The R-IN32M3 is commercially available and the libraries for R-IN32M3, such as the RTOS driver, and protocol stacks, are available from Renesas Electronics web site.

The R-IN32M3 provides various types of communication ports, such as CAN and CC-Link, while two Ethernet ports are provided for use by industrial Ethernet systems.

The R-IN Engine is a module that implements industrial network functions. It is not a conventional simple processing unit but it provides high performance and has high added value, as follows.

1) LC-HWRTOS: The LC-HWRTOS comprises the AAI, the ARTESSO HWRTOS, and the core, as mentioned in section III. A Cortex-M3 is adopted as an ARM core.

2) 802.3 Two-port Switch: The module operates with a daisy chain configuration using a two-port PHY in industrial Ethernet. Two different hardware configurations are possible, EtherCAT/slave and CC-Link IE/Field.

3) Ethernet Accelerator: The Ethernet Accelerator has three functions for accelerating protocol processing. (i) Checksum execution for TCP and IP. (ii) A protocol header rearrangement function, which rearranges the compressed header format into a format that the core can handle easily, and vice versa. (iii) A buffer management function that comprises buffer allocation and release functions. They are also implemented by hardware logic.

V. Performance Evaluations

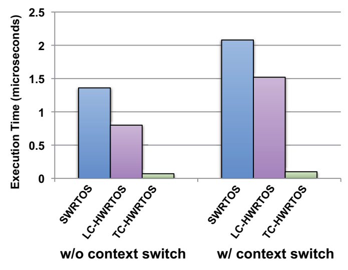
This section presents comparisons of the performance obtained using TC-HWRTOS, LC-HWRTOS, and SWRTOS.

A. *Evaluation Items*

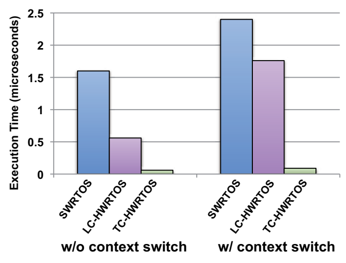
The RTOS performance and network performance are evaluated. *1)* to *5)* present evaluations of the RTOS performance and *6)* describes the network performance, as follows.

1) API execution time: The improvement in the execution time with HWRTOS is evaluated. The API execution time is measured with and without context switching for each of the ”start task,” ”release semaphore” and ”wake up task” APIs.

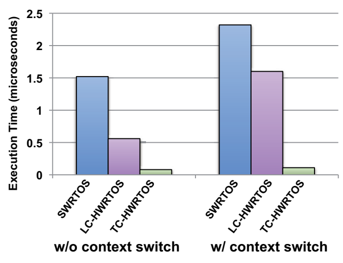
2) Interrupt response: The improvement in the interrupt response with the IIA offloading is evaluated. The times are



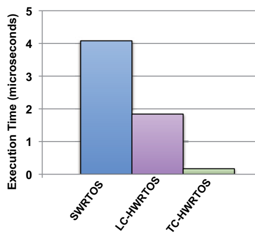
1)-a **Start task API execution**



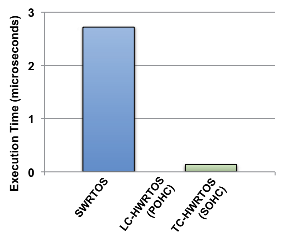
1)-b **Wake up task API execution**



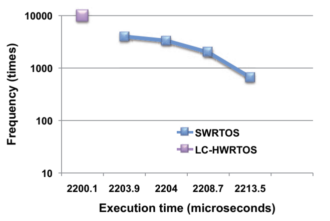
1)-c **Release semaphore API execution**



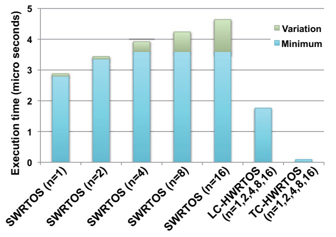
2) **Interrupt response**



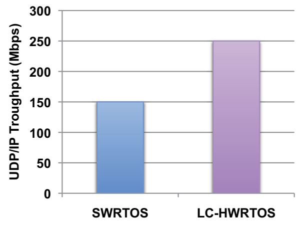
3) **Influence of interrupts**



4) **Influence of Tick**



5) **Wake-up execution time**



6) **UDP/IP Throughput**

Fig. 8. **Evaluation Result**

measured from t1 to t3 in Fig.10 (1) with SWRTOS, in Fig.11 with TC-HWRTOS and in Fig.5 with LC-HWRTOS.

3) IIA overheads: The improvement in IIA offloading with the POHC is evaluated. In this case, the task is not switched as the result of the API execution, which is invoked during the IIA. The times are measured from t1 to t2 in Fig.10 (1) for SWRTOS, and in Fig.11 for TC-HWRTOS. In LC-HWRTOS, the time is zero is verified.

4) Influence of tick: The improvement in the real-time response with the tick management offloading is evaluated, i.e., the variation in the interrupt response with the tick function. Three periodic tasks are performed, where each cycle is set to 4, 10, or 15 ms. Each task process executes 20,000 loops, which execution time is about 2.2 ms. As shown in Fig.7, the execution time of the 4-ms periodic task is measured. If there are no influences on the tick, the execution time is about 2.2 ms and fixed.

5) Wake-up execution time: The improvement in the real-time performance in hardware implementation is evaluated. The execution time of the API is measured if n tasks are waiting for their timeout in the queue and one of them is woken by the wake-up task API. The measurements are executed 1 million times for each n, and the maximum and minimum times for each n are obtained.

6) UDP/IP throughput: The UDP/IP throughput is evaluated using each LC-HWRTOS and SWRTOS.

B. Experimental Setup

The performance is evaluated with SWRTOS, LC-HWRTOS, and TC-HWRTOS which are API-compatible. With SWRTOS and the LC-HWRTOS, the R-IN32M3 evaluation board is used. The operational clock of both the core and the HWRTOS are set to 100 MHz. In the SWRTOS evaluation, the HWRTOS function of the R-IN32M3 is disabled. In the TC-HWRTOS, the Verilog source code of the original ARTESSO system shown in the Appendix is used and the time is obtained by the Verilog RTL simulator.

In the network performance evaluation related to the SWRTOS and the LC-HWRTOS, the evaluation environment described above is employed and the UDP/IP protocol stack is implemented on it, where the UDP/IP throughput is measured in both environments.

C. Evaluation Results

The following section presents the evaluation results for the experiments described in the previous section.

For *1)*, Fig.8 1) a–c show the execution time results with the three APIs. Without context switching, the execution times with the LC-HWRTOS are 1.7 to 2.9 times faster compared with the SWRTOS. With context switching, however, the execution times with the LC-HWRTOS are 1.4 to 1.5 times faster compared with the SWRTOS. The results are better ”without context switching” with the LC-HWRTOS because only pre- and post-API software processing are needed in ”without” case, but context-switching software is added process in ”with” case. As mentioned in Section II, execution time in the TC-HWRTOS is the fastest.

For *2)*, Fig.8 2) shows the results of the interrupt response evaluation. The execution time with the LC-HWRTOS is 2.3 times faster compared with that with the SWRTOS.

For *3)*, Fig.8 3) shows the IIA overheads. With the LC-HWRTOS, the IIA function is offloaded from the core and context switch is not executed thus the IIA overheads are zero. The results demonstrate the system performance is improved by POHC method, because the core can keep to execute in case of a context switch is not needed, as shown in Fig.5.

For *4)*, Fig.8 4) shows the effects of the time tick management function on real-time processing. The evaluation is only executed in the SWRTOS and the LC-HWRTOS because the performance of TC-HWRTOS is almost same as that of

LC-HWRTOS. The horizontal axis indicates the execution time required for the 4-ms periodic task. The execution time required for the 4-ms periodic task in the SWRTOS varies according to the periodic interrupt and the maximum range of variation is 9.6 ms. By contrast, the execution time in the LC-HWRTOS is always 2.2001 ms. These results show that real-time processing is delayed by a maximum of 9.6 ms by tick management function with the SWRTOS, whereas the process is not delayed with the LC-HWRTOS. Therefore, the LC-HWRTOS is advantageous during real-time processing because a task that is invoked by an interrupt can start at a precise time.

For *5)*, Fig.8 5) shows the execution time of the wake up task when some tasks are waiting for timeouts. With the SWRTOS, the maximum execution time and its variation increase according to n. When n is 16, the variation is 1.04 μs. By contrast, the execution time is always fixed regardless of n with the LC-HWRTOS. The execution time with the LC-HWRTOS is 2.6 times faster compared with the SWRTOS when n is 16. These results show that the wake up task execution time varies according to the internal conditions in the SWRTOS, whereas the time does not depend on the internal conditions in the LC-HWRTOS. Therefore, the LC-HWRTOS is advantageous for real-time processing because a task can wake up at a precise time in the LC-HWRTOS.

For *6)*, Fig.8 6) shows the UDP/IP throughput results where both platforms are exactly the same, except for the RTOS. The results indicate that the performance is improved by 1.67 times only when the SWRTOS is replaced by the LC-HWRTOS. The results also show that the network load decreased by 40% with the LC-HWRTOS, thus LC-HWRTOS would be effective in industrial network systems.

VI. Related Work

Various techniques have been proposed for improving the performance of RTOSs, some of which implement the RTOS functions partially in hardware [6-11]. Others implement all of the functions of the RTOS in hardware [1-5]. Previous studies have shown that the performance of the HWRTOS was several times faster than that of the SWRTOS. Some of them adopt ARM core, however, they implemented non-standardized and limited number of APIs, and they only provided several or several tens of queues because they could not implement a large number of queues in small volume hardware, therefore insufficient objects could be provided. Consequently, their methods did not satisfy the requirements of industrial network systems. Furthermore, they have not been commercially available as a SoC. By contrast, the proposed architecture provides 41 ITRON[13] standard APIs, which are sufficient for utilization in the IndCntlrs. ITRON is a RTOS standard and widely used in Japan. The proposed architecture also provides several thousand queues at low cost using the novel Virtual Queue technology, as described in the Appendix. Therefore, the proposed architecture satisfies the requirements of industrial network systems. The SoC based on the proposed architecture is the world's first commercial product which implements ARM and RTOS in hardware.

VII. Conclusion

Recently, faster network systems have been deployed for industrial networks using Ethernet. However, the increased protocol processing load affects real-time processes such as motor control. Thus, we developed R-IN32M3 SoC to overcome this problem, which can be used in industrial network systems. The requirements of industrial network systems are: (1) reducing the load of network protocol processing, (2) improved real-time capability, (3) multi-protocol support, (4) adoption of ARM cores, and (5) low costs and low power consumption. To satisfy these requirements, Corex-M3 was adopted as the core and the LC-HWRTOS was adopted as the RTOS. Our evaluations showed that the LC-HWRTOS operated faster than the SWRTOS. Our experimental results showed that the UDP/IP throughput was increased by 1.67 times by replacing the SWRTOS with the LC-HWRTOS and the network load decreased by 40%.

References

[1] Lindh L., "Fastchart – a fast time deterministic CPU and hardware based real-time kernel," in Proc. of Euromicro Workshop on Real Time Systems, pp. 36–40, Jun, 1991

[2] Adomat J., Furunas J., Lindh L., Starner J., "Real-time kernel in hardware RTU: a step towards deterministic and high-performance real-time systems," in Proc. of the 8th Euromicro Workshop, pp. 164–168, Jun 1996

[3] Nordstrom S., Lindh L., Johansson L., Skoglund T., "Application specific real-time microkernel in hardware," in Proc. of Real Time Conference, 2005.

[4] Samuelsson T., Åkerholm M., Nygren P., Johan Stärner J., Lindh L., "Comparison of multiprocessor real-time operating systems implemented in hardware and software," in Proc. of Int'l Workshop on Advanced Real-Time Operating System Services (ARTOSS'03), 2003.

[5] Nakano T., Utama A., Itabashi M., Shiomi A., Imai M., "Hardware implementation of a real-time operating system," in Proc. of 12th TRON Project International Symposium (TORN'95), pp. 34044, 1995.

[6] Kohout P., Ganesh B., Jacob B., "Hardware support for real-time operating systems," in Proc. of the 1st International Conference on Hardware/Software Codesign and System Synthesis, pp. 45–51, Oct. 2003

[7] Chandra S., Regazzoni F., Lajolo M., "Hardware/software partitioning of operating systems: a behavioral synthesis approach," in Proc. of the 16th ACM Great Lakes Symposium on VLSI, pp. 324–329, 2006

[8] Parisoto A., Souza A. Jr, Carro L., Pontremoli M., Pereira C., Suzim A., "F-Timer: dedicated FPGA to real-time systems design support," in Proc. of 9th Euromicro Workshop on Real-Time Systems, pp. 35–40, 1997

[9] Mooney III V., Lee J., Daleby A., Ingstrom K., Klevin T., Lindth L., "A comparison of the RTU hardware RTOS with a hardware/software RTOS," in Proc. of Design Automation Conference, 2003, pp. 683–688.

[10] Mooney III V.J., Blough D.M., "A hardware-software real-time operating system framework for SoCs," IEEE Design & Test of Computers, 44–51, 2002.

[11] Mooney III. V., "Hardware/software partitioning of operating systems," in Proc. of Design Automation and Test in Europe Conference (DATE'03), 2003, pp. 338–339.

[12] Maruyama N., Ishihara T, Yasuura H., "An RTOS in hardware for energy efficient software-based TCP/IP," Proc. of IEEE Symposium on Application Specific Processors (SASP), 2010, pp. 13–18.

[13] TRON ASSOCIATION, "μITRON4.0 Specification," 1999.

[14] Felser M., "Real-time Ethernet – industry prospective," Proceedings of the IEEE, Volume 93, Issue 6, June 2005, pp. 1118–1129.

[15] Renesas Electronics, "R-IN32M3-Series Data Sheet," Dec 9, 2013.

Appendix

This appendix describes the original ARTESSO system [12], which we developed using the ARTESSO HWRTOS, as shown in Fig.9.

A. *Summary of the original ARTESSO system*

The original ARTESSO system comprises an ARTESSO HWRTOS, a Register Save Memory, and an ARTESSO core, as shown in Fig.9. They are configured as a TC-HWRTOS. The ARTESSO HWRTOS conforms with ITRON specifications [13] and it supports 41 ITRON APIs. The supported APIs are listed in Table II. The Register Save Memory is used to maintain the contents of the ProcReg on a task-by-task basis. The ARTESSO Core is a proprietary 32-bit RISC processor, which has a special interface that connects with the ARTESSO HWRTOS in a tightly coupled manner.

B. *Features of the original ARTESSO system*

The following are the features of the original ARTESSO system.

1. Configuration of TC-HWRTOS.
2. Several thousand queues in hardware at a low cost based on an innovative idea called Virtual Queue.
3. An interrupt invoked API (IIA) offloading function that executes interrupt processing in hardware, which improves the performance of the interrupt response.
4. A tick management offloading function that removes the software tick process, which also improves the interrupt response.

1) TC-HWRTOS: Fig.9 shows the architecture of the original ARTESSO system, which is configured as a TC-HWRTOS. The Main Controller (MC) is implemented by a hardware state machine that executes all of the API call processes. The Task Control Block maintains the management information related to all of the tasks, such as the ”current state” and ”task priority.” The Virtual Queue module implements all of the queues used by the RTOS. The Event Control Block, the Semaphore Control Block, and the Mail Box Control Block maintain and manage the information required by event functions, semaphore functions, and mailbox functions, respectively.

Next, we describe the API call procedure. The arguments and return values of the APIs are handed over using some of the general purpose registers in the ProcReg. When the core decodes an API call assembler instruction, the core changes the Req_API signal to 1 and indicates an API identifier by sending the API_ID signal to the MC. If the MC detects this signal, it changes the hold signal to 1 and commences API processing according to the API_ID signal. If the API has arguments, the MC refers to the specific registers in the ProcReg that are assigned to the arguments. When the MC completes the execution of the API process, it writes the return value in the specific register of the ProcReg that is assigned to the return value and it then changes the hold signal from 1 to 0, thereby indicating the completion of the process to the core.

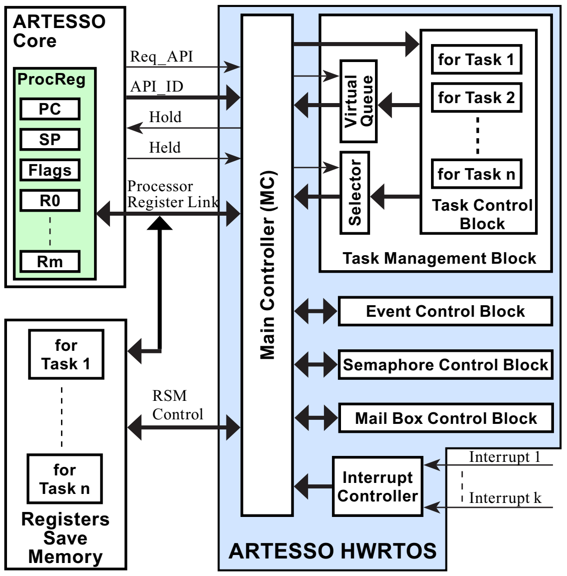


Fig. 9. **Original ARTESSO system (TC-HWRTOS)**

TABLE II
Supported APIs

task	start, exit, terminate, change priority, get task ID, sleep, wakeup, release wait
flag	create, delete, wait, set, clear, poll
semaphore	create, delete, wait, release
mailbox	create, delete, send, receive
cpu	lock, unlock
dispatch	disable, enable
ready queue	rotate
system timer	set, get

If the core recognizes the hold signal transition, it restarts and fetches from the next program counter.

If a context switch is needed after the completion of API execution, the MC saves the contents of the ProcReg in the Register Save Memory and loads the next task contents into the ProcReg from the Register Save Memory. Next, the MC changes the Hold signal from 1 to 0, which indicates completion to the core and the core then restarts the fetch process again.

In the interrupt procedure, interrupt signals enter the Interrupt Controller in the ARTESSO HWRTOS. If the MC detects an interrupt through the Interrupt Controller, the MC changes the Hold signal to 1 to stop the core. When the core stops, the Held signal is changed to 1. If the MC detects the Held signal transition, it saves the contents of the ProcReg to the Register Save Memory and loads the ISR contents from the Registers Save Memory into the ProcReg. Next, the MC changes the Hold signal from 1 to 0 to indicate the completion of the context switch to the core and the then core restarts the fetch process according to the ISR register set.

As mentioned above, the original ARTESSO system implements the TC-HWRTOS using a special interface with a procedure between the ARTESSO HWRTOS and the ARTESSO

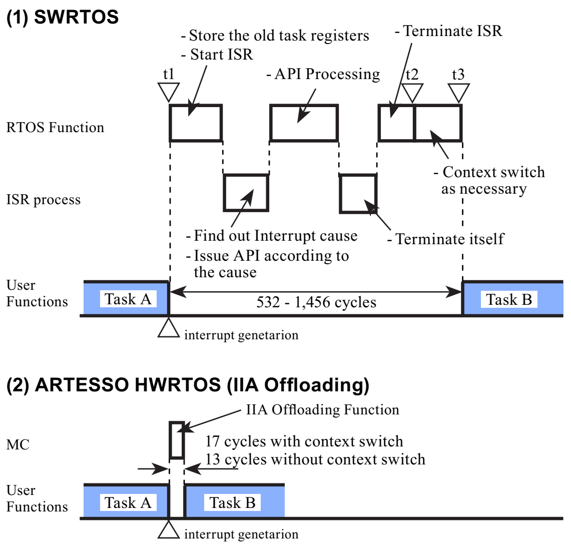


Fig. 10. **ISR and IIA operations**

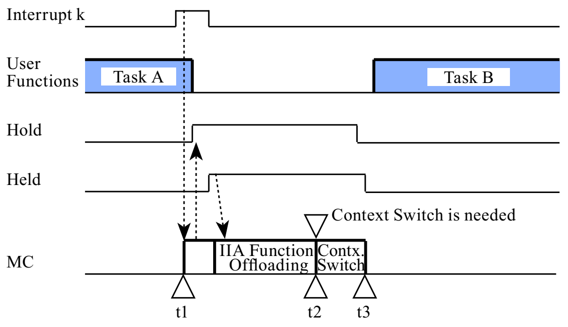


Fig. 11. **IIA offload execution**

core.

2) Virtual Queue: In general, RTOSs require many queues, i.e., several thousand queues, since they are needed for each instance of an object. To implement an RTOS in hardware, the queues should also be implemented in hardware. However, it is very expensive to implement many queues in hardware using conventional technology such as hardware FIFOs. Thus, we propose the innovative idea of a Virtual Queue, which is an information compression technique with reversibility that produces a large number of queues with a small hardware volume.

3) IIA Offloading: The IIA offloading function is implemented in addition to the conventional interrupt service routine (ISR) in the ARTESSO HWRTOS. The IIA offloading function invokes a selected API based on the cause of the interrupt. Fig.10 (1) shows that the SWRTOS executes the same procedure as IIA offloading. The following description provides specific details of the procedure. When an interrupt is generated, the SWRTOS starts and it saves the current task register contents in the ProcReg, before loading the ISR register contents into the ProcReg and starting the ISR. The ISR determines the cause of the interrupt and invokes an API based on the cause. After the completion of API execution, the SWRTOS terminates the ISR and the SWRTOS then replaces the ISR contents in the ProcReg with the new task register contents.

IIA offloading can replace the software ISR process, as mentioned above, therefore all of the interrupt processes can be implemented in hardware. Thus, the interrupt response is improved dramatically, as shown in Fig.10 (2) and Fig.11.

4) Tick management Offloading: The SWRTOS uses a tick process to implement timeout processing for tasks. The tick process is woken up by a periodic interrupt and it decrements each timeout counter. If it detects a timeout, it removes the task from the wait queue and appends it to the ready queue. As mentioned above, the tick process is a critical process and interrupts are inhibited during the process, which results in fluctuations in the interrupt latency. By contrast, the ARTESSO HWRTOS implements hardware timeout counters on a task-by-task basis and the counter value is decremented by its hardware. If the counter reaches zero, the MC is started, which removes the task from the wait queue and appends the task to the ready queue. Next, the MC executes context switching if necessary. Thus, the ARTESSO HWRTOS does not require the tick process in software. This results in a drastic reduction in fluctuations in the interrupt latency.

TABLE III
RTOS execution time

System Call	Dispatch	SWRTOS	ARTESSO HWRTOS (TC-HWRTOS)
Sleep Task	Yes	628	10
Wakeup Task	Yes	496	10
Change Priority	Yes	541	11
Receive from Mailbox	No	224	7
Receive from Mailbox	Yes	591	11
Send to Mailbox	No	360	8
Send to Mailbox	Yes	541	11
Wait Semaphore	No	216	6
Wait Semaphore	Yes	558	9
Release Semaphore	No	344	7
Release Semaphore	Yes	536	11

Unit : Cycles

C. Performance of the original ARTESSO system

Table III compares the performance of commercial SWRTOS and the ARTESSO HWRTOS, which has the TC-HWRTOS configuration. The RTOS performance of the original ARTESSO system was several tens times faster than that of SWRTOS. The interrupt response using IIA offloading was 38 to 104 times faster than that of the SWRTOS, as shown in Fig.10.

Distributed Real-Time Fault Tolerance on a Virtualized Multi-Core System

Eric Missimer*, Richard West and Ye Li

Computer Science Department
Boston University
Boston, MA 02215
*Email: {missimer,richwest,liye}@cs.bu.edu *VMware, Inc.*

***Abstract*—This paper presents different approaches for real-time fault tolerance using redundancy methods for multi-core systems. Using hardware virtualization, a *distributed system on a chip* is created, where the cores are isolated from one another except through explicit communication channels. Using this system architecture, redundant tasks that would typically be run on separate processors can be consolidated onto a single multi-core processor while still maintaining high confidence of system reliability. A multi-core chip-level distributed system could therefore offer an alternative to traditional automotive systems, for example, which typically use a controller area network such as CAN bus to interconnect multiple electronic control units. Using memory as the explicit communication channel, new recovery techniques that require higher bandwidths and lower latencies than those of traditional networks, now become viable. In this work, we discuss several such techniques we are considering in our chip-level distributed system called Quest-V.**

I. Introduction

Fault-tolerance in real-time systems has historically been accomplished through redundancy in both hardware and software [1]–[4]. For example Briere et al. explain how in the Airbus A320/330/340 there are "sufficient redundancies to provide the nominal performance and safety levels with one failed computer, while it is still possible to fly the aircraft safely with one single computer active [5]." Redundancy across multiple compute nodes protects against both hardware and software failures. We propose a technique that uses hardware virtualization to sandbox each core of a multicore processor, along with a subset of machine physical memory and a collection of I/O devices.

This design allows the system to operate as a *separation kernel* [6], where communication between sandboxes occurs through explicit memory channels and inter-processor interrupts. Each sandbox is isolated from the software faults that can occur in another sandbox and any hardware faults that are specific to a single core or subset of memory. This allows the consolidation of computational tasks onto a single multi-core processor while still maintaining a large number of the advantages associated with a traditional distributed system. This approach also has the advantage of having a higher bandwidth communication channel, i.e. memory, than in traditional control networks such as CAN, enabling new recovery techniques. In this paper we focus specifically on different Triple Modular Redundancy [7], [8] (TMR) techniques that can be applied to a virtualized distributed system on a chip.

The next section provides a brief summary of Quest-V, a separation kernel we have been developing for use in real-time mixed-criticality systems [9]. Section III discusses our generic fault *detection* approach. Section IV introduces three fault *tolerance* methods we are currently considering for Quest-V, and each approach is described in detail in Sections V- VII. This is followed by a discussion of our generic fault *recovery* technique in Section VIII. Related work is discussed in Section IX followed by the conclusions and future work in Section X.

II. Quest-V Design

Quest-V is designed for real-time high confidence systems. It is written from scratch for Intel's x86 architecture, and operates as a *distributed system on a chip*. Quest-V uses hardware virtualization technology to partition a one or more cores, a region of machine physical memory and a subset of I/O devices into separate *sandboxes*. Intel's extended page table (EPT) feature on modern VT-x processors is used to partition memory into separate regions for different sandboxes.[1]

Hardware virtualization assists in the design of virtual machine monitors (VMMs, a.k.a. hypervisors), which are capable of managing multiple virtual machines (VMs) on the same physical machine. Unlike in traditional VMMs, Quest-V does not require the use of a monitor to schedule VMs on processor cores. Instead, each sandbox runs its own kernel that performs (sandbox-local) scheduling directly on available processor cores.

[1]AMD x86_64 and ARM Cortex hardware virtualization extensions provide a similar functionality.

This approach has numerous advantages. First, because a trusted monitor is not performing VM scheduling, it is eliminated from the normal operation of the system. This results in very few VM exits, resulting in lower overheads due to virtualization. Second, this simplifies the design of a monitor, which now only needs to initialize sandboxes (or, equivalently, VMs) [2], establish shared memory communication channels between sandboxes, and assist when necessary in fault recovery. The reduced complexity of Quest-V's monitor code makes it more easily verified for correctness.

III. Fault Detection

Before we discuss different possible TMR configurations in Quest-V, we will briefly discuss how we detect faults. This approach is used throughout all the subsequently described system configurations. TMR uses a majority voting mechanism to detect an error. In the traditional TMR setup, the results of redundant (replicated) computations are sent to the voter. While we could use this approach, we have decided to use a more aggressive fault detection technique that can detect any deviation in the redundant computation, not only the result. Our approach takes hashes of memory on a per-page basis of all memory modified by the program between synchronization points. We also take a summary hash of all the modified memory. The voter first compares the summary hashes. If the summary hash values are identical, no further error detection actions are taken. If the summary hashes are not identical, the voter iterates through the per-page hashes to determine which pages are different. This allows for faster recovery, described in detail in Section VIII. Currently, we are using the Jenkins one-at-a-time [10] hash function due to its simplicity. For added security, where a malicious sandbox could corrupt a page so that the hash would be the same as the valid page, a cryptographically secure hash function could be used.

Taking consensus on memory hashes is a generic fault detection approach as it just relies on the program's user-space state or the entire sandbox being identical across all instances. This also places a restriction on the types of programs that can be monitored. Their execution across sandboxes must be identical during a fault-free execution. The task or guest can only rely on its own internal state and any data passed to the program through shared memory or by the hypervisor. Redundant tasks or guest code must not make use of process-specific values such as process IDs, unless they are identically replicated, and similarly should not use constructs such as `gettimeofday()` which might differ across replicas. We do not believe this restriction is too prohibitive; however, we plan to remove this restriction in future work.

[2]Initializing a sandbox requires setting up an appropriate hardware partition of processor cores, memory and I/O devices. Devices themselves can be shared but resource partitioning is typically done only once, at boot-time.

During execution, the redundant guests or task instances reach specific synchronization points. Depending on whether the redundancy is for the entire sandbox (guest) or only a single task determines whether the synchronization points are when a VM exit occurs or when the replicated task makes a system call. This influences whether synchronization is handled in the hypervisor or a guest kernel. At the synchronization point, the hypervisor or kernel determines which pages have been modified and creates the necessary per-page hashes and summary hash. The memory management unit can be used to help track which pages have been modified as the pages can initially be marked as read only to cause hypervisor or kernel traps on attempted page updates. The hashes are sent to the voter and the sandbox/task continues execution. We do not halt execution, i.e. barrier synchronize the tasks, and wait for the results of the voter as this would add unnecessary overheads. Instead, no external action, e.g. output to an actuator, is taken on behalf of the redundant copies until a majority of identical hashes have been collected. Once enough hashes have been collected and verified, any necessary I/O is performed and the hashes are released.

If an error has occurred but there is still a majority consensus, any I/O is performed, and the hypervisor or kernel is notified as part of fault recovery (see Section VIII). If there is no consensus, an application dependent recovery procedure can be used to bring the system into a useful state. For example, for an autonomous automobile application, the system could safely bring the vehicle to a halt. To detect performance faults, where a sandbox or task does not reach a synchronization point, timeout values can be specified by the application developer. If a synchronization point is not reached within the timeout value it is treated as a fault and the same generic recover procedure is used to correct the performance failure.

IV. Virtualization Based Triple Modular Redundancy

The chip-level distributed system design of Quest-V creates an opportunity to develop new fault tolerance techniques. For example, techniques that exploit high bandwidth, low latency communication between sandboxes can be explored. In this section, we will introduce various techniques and we will highlight their strengths and weaknesses in Sections V, VI and VII.

Redundancy in either data and/or execution can be used to detect Byzantine errors, e.g. soft errors causing bit-flips, possibly as a result of the system being exposed to radiation. Separate Quest-V sandboxes can support redundant executions of a process or guest. Whenever an external action needs to be taken, e.g. sending a message to an actuator, a consensus mechanism, such as Triple Modular Redundancy [7], [8] (TMR), can be used to ensure that faulty values do not propagate to the device. We will focus our discussion of fault tolerance techniques in Quest-V on those that follow the TMR approach. In what follows, we describe three different fault tolerant Quest-V system configurations, which depend on where the voting mechanism and device driver are located:

- **Voting mechanism and device driver in the hypervisor** – Each process submits its results to the hypervisor via a hypercall or through emulated devices. The hypervisor waits for the results or a timeout, compares the results and sends the majority to the device.
- **Voting mechanism and device driver in one sandbox** – A single sandbox acting as an *arbitrator* has sole access to the device. The arbitrator is responsible for comparing the results of redundant computations, which are distributed across the other *computation* sandboxes. Communication between the arbitrator and each computation sandbox is via a separate shared memory channel, which is protected by extended page table (EPT) mappings. That is, no two computation sandboxes can access each other's private channel to the arbitrator, thereby preventing a faulty computation sandbox from corrupting the results of another sandbox.
- **Voting mechanism distributed across sandboxes and device driver is shared** – Each sandbox that contains a redundant process also has shared access to the device. Each redundant process compares its own state to the redundant copies to determine if a fault has occurred and recovers if necessary. The non-faulty sandboxes elect a leader to send the data to the device. Communication occurs in pair-wised private shared memory channels as described in the previous configuration.

In the following three sections, we discuss further details of our proposed approach for each of the above configurations.

A. *Voter – Single Point of Failure*

The final voter in a TMR setup can be a single point of failure. The voter could malfunction and select the minority result or simply output a different result than its inputs. All the configurations above could suffer from the voter malfunctioning. A common solution is to use three voters [11] and the output of each voter is the input to the next stage of the computation. However, eventually a single output that is to be sent to the device needs to be determined (assuming the device does not support three results and will perform its own TMR). Techniques have been developed to protect single points of failure such as voters. For example, Ulbrich et. al used arithmetic encoding techniques to protect the voters in TMR [12]. Furthermore, while the redundant sandboxes might not have direct access to the device in the first and second configurations described above, they could have read-only access to ensure that the voting mechanism is behaving correctly. If the redundant sandboxes reach a consensus that the voting mechanism is behaving incorrectly the device driver and voter could be re-initialized or the device could be mapped to a new sandbox. We plan to explore these techniques in future work.

V. Hypervisor Voting and I/O

In this configuration, the voting mechanism and device driver are located in the hypervisor. This conflicts with the design philosophy of Quest-V, as the hypervisor should ideally be as simple as possible. Furthermore, placing a device driver in the hypervisor could make the entire system vulnerable if the device driver is incorrectly implemented. However, if we overlook this, TMR fault tolerance can be applied to operating systems other than Quest-V, e.g. Linux, without the need to modify any source code. Specifically, the hypervisor could host three or more redundant guests that communicate the results of the computation through an emulated I/O device. Besides voting and performing I/O, the hypervisor is responsible for encapsulating each guest to ensure that they remain in loose lockstep. This approach is depicted in Figure 1.

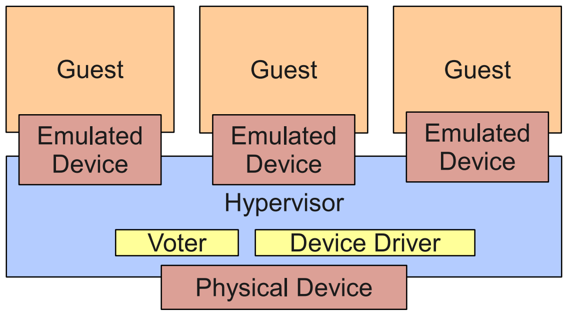


Fig. 1: Voting mechanism and device driver in the hypervisor.

This approach builds upon hypervisor-based fault tolerance (HBFT) [13]–[16]. HBFT is an example of a primary-backup based fault tolerance system that uses a hypervisor to encapsulate a guest virtual machine so the state of the entire system can be easily manipulated and saved. This allows the primary and backup, which run on different machines, to be easily synchronized without relying on any information from, or modification of, the operating system or application. The execution of the guests is divided into time epochs. At each epoch, the primary and backup are ensured that they are in the same state. This is either accomplished by having an active backup that is kept in lockstep [13], having a dormant backup and transferring the changed state from the primary to the backup [14], or having an active backup that is not kept in lockstep and any discrepancies between the two executions are handled at the epoch boundaries [15]. During execution, the hypervisor buffers all output until an epoch has been reached and the primary and backup have been synchronized. This ensures that if the primary fails, the backup can perform the same output without duplicated output being seen outside of the system.

HBFT, as it stands, can only handle crash failures, where the primary node halts when it fails [17], [18]. If the primary fails, the backup guest begins execution to ensure high availability. The HBFT model can be extended to recover from Byzantine errors by adding a third redundant guest. At epoch boundaries, i.e. synchronization points, the hypervisor examines the state of each guest and if a single guest differs, the state can be corrected. This approach would not require any modification of the guest operating system or application and could even be applied to closed-source software. By combining all the guests onto a single multi-core processor, we cannot recover from errors such as power failures that bring down the entire processor, but we can recover from Byzantine errors in a much more efficient manner as data can be transferred between guests at a much faster rate.

This approach does have some disadvantages compared to the other two approaches. First, as previously mentioned, the hypervisor must be much more advanced. For example, it must support emulated devices and be able to recover the entire guest operating system. Also, every sandbox must be performing the exact same operations. It is not possible for one sandbox to run non-safety critical tasks that the other sandboxes do not run as the hypervisor has no information about what state belongs to which task. We will see in the next two sections that this limitation does not apply to the other approaches.

VI. Arbitrator Sandbox

In this configuration, the voting mechanism and device driver are located in an *arbitrator* sandbox. The redundant computations are performed in three or more guests as depicted in Figure 2. Communication between guests is explicit through shared memory channels. These could be set up statically by the hypervisor at boot-time, or dynamically, at run-time. Unlike the approach described in Section V, the fault tolerance is at the application-level as opposed to the entire guest. This has the advantage that each guest does not need to be identical, e.g. only a subset of the applications running in each guest need to be replicated. Applications that are not safety-critical can be executed in just one guest. This approach requires operating system support, to hash the application pages and communicate the results to the arbitrator sandbox.

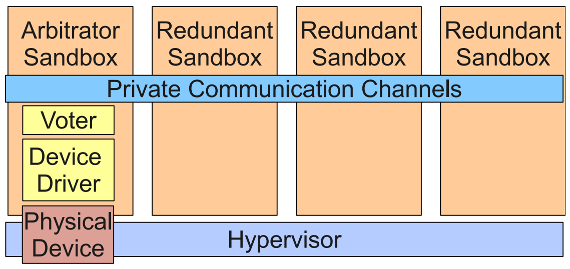


Fig. 2: Voting mechanism and device driver in an arbitrator sandbox.

The main advantage to this approach is that the hypervisor remains as simple as possible. The hypervisor does not have to emulate any devices, nor does it contain any device drivers. All necessary devices are isolated from the redundant guests via hardware virtualization extensions, as described in our earlier work [9]. Not only is the hypervisor simple but it is kept out of the control path during normal execution, thus avoiding the costs of VM exits. Only during recovery would it be necessary to drop down into the hypervisor. Also, as previously stated, the granularity of redundancy is much finer as the redundancy is at the application-level as opposed to the guest-level.

Having a single sandbox vote and send the results to the device driver does have its limitations. First, a sandbox is necessary to contain the voting task. While the task performing the voting would require a low overall utilization, the other approaches do not require a separate sandbox just for voting. Another limitation is that while we gain the ability to do task-level redundancy we do so at the cost of having to add the redundancy support into the operating system. While this is possible for an operating system such as Quest-V, it becomes increasingly difficult for a more complex system such as Linux.

VII. Shared Voting

A third configuration is to have the voting process and device shared across all sandboxes. This approach is similar to the second approach in that it has a smaller granularity of redundancy, e.g. at the application level, and therefore the redundant sandboxes do not have to be executing identical sets of tasks or operate in lockstep. It also requires operating system support to hash memory pages and communicate the results to different sandboxes. This approach avoids the need for a special arbitrator sandbox. Each sandbox takes a vote on the results of the other sandboxes, again communicating through private shared memory channels. Of the sandboxes that have a value equal to the majority vote, a temporary leader is elected, which performs the actual I/O. This approach is depicted in Figure 3.

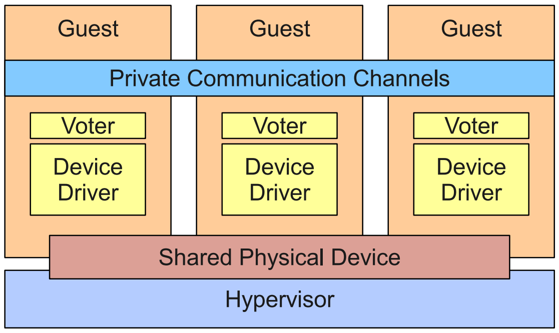


Fig. 3: Voting mechanism and device driver are shared across sandboxes.

There are two possible ways that the shared driver could be implemented. One is that the device is directly mapped to all participating sandboxes all the time. This has obvious security and safety concerns as a faulty sandbox could at any time send erroneous output signals through the device. As previously mentioned in Section IV, the other sandboxes could be monitoring the memory associated with the device to determine if such behavior occurs and if so, signal to the hypervisor that the device should be unmapped from the faulty sandbox. This could be acceptable if a few erroneous I/O messages were permitted and the device could be reinitialized.

A more secure method would be the following: whenever an I/O operation needed to be performed, each sandbox sends to the hypervisor the sandbox it wishes to be the leader to perform the I/O operations. The hypervisor then temporarily grants that sandbox the write privileges to the device while still permitting the other sandboxes to have read access. Once the I/O operation is complete the sandboxes signal that the write privileges should be revoked. The granting and revoking of write privileges performed by the hypervisor only occurs if a majority of the sandboxes signal that it should occur. In this way, a sandbox would have to fail specifically after it was granted write privileges and before the privileges were revoked. Such an approach requires some support from the hypervisor as the hypervisor has to be able to dynamically map and unmap a device to different sandboxes. It would also involve a large number of VM exits which would be required to temporarily grant and then later remove access to the device. Also, how interrupt service routines should be handled is unclear as they often require direct access to the device and occur asynchronously.

VIII. Recovery

We have a few key requirements for our recovery procedure. First, it should be as generic as possible, so that it can be used across multiple applications. Second, the recovery procedure should be applicable to both an entire operating system running within a sandbox and to a task running within an operating system. Obviously there will be some differences, mostly complications due to operating system recovery, but the general approach should be the same. This allows us to share a similar code base between operating system and task recovery, reducing the code base size and the possibility of errors.

Initially, our recovery procedure involved taking snapshots of the changed state at each synchronization point, similar to a rollback procedure. However, instead of rolling back to the last known good state we would instead roll forward using the snapshot of a correct instance to bring an incorrect instance up to the same state. However, in our preliminary evaluations, the snapshot procedure dominated the overhead associated with recovery to the point that rolling back and rolling forward had nearly identical recovery times. We therefore decided to abandon taking snapshots at synchronization points and perform recovery without them.

The point of taking snapshots was to allow one sandbox or task to be recovered without interfering with the execution of the sandbox that is being used as the correct instance. The snapshot pages could be read without the fear that the correct instance would modify them while the recovery procedure was occurring, as the correct instance would actually be using different memory frames at the time of recovery. If we do not take snapshots, then we run the risk that the correct instance modifies a page while we are copying it for recovery. This is the same issue that occurs during live migrations of virtual machines [19]. The solution is to divide the migration, or in our case, the recovery procedure into different phases. First, pages are pushed from source (correct instance) to destination (recovering instance), and if a page is modified, it is re-pushed. Second, the source is stopped and pages are copied without the need to be concerned about consistency. Finally, as the migrated virtual machine executes, any pages that have not been pushed are pulled as they become necessary. Different live migration strategies balance these phases. We can use a similar approach to recovery which basically involves performing a live migration on a sandbox or application. However, instead of halting the source instance, we allow it to continue running. We will explore what balance of the three phases is most appropriate for a recovery.

IX. Related Work

Achieving triple modular redundancy through multiple executions of a task has been previously explored by Reinhardt and Mukherjee [20], and Döbel, Härtig and Engel [21]. Reinhardt and Mukherjee developed a simultaneous and redundantly threaded processor, in which hardware is responsible for error checking and recovery. While this alleviates software developers of fault tolerance concerns it also adds extra overhead by replicating components that are not safety-critical. Furthermore, specialized hardware features must be available.

Döbel et al. developed a software based approach to task replication on top of the L4 Fiasco microkernel [21]. Their approach involved a master-controller task, which monitored the execution of redundant tasks. The controller handled CPU exceptions, page faults and system calls made by the redundant tasks and ensured they had identical state at these points. This is similar to our first approach of using a hypervisor as the master controller. As with our other two approaches, it operates on a per-task basis rather than an entire guest.

X. Conclusions and Future Work

We have presented three fault tolerant configurations based on TMR. Such techniques are made possible by the unique design of the Quest-V separation kernel. We have focused on TMR as it is a common fault tolerance mechanism used to handle soft errors.

Beyond implementing and comparing the previously described techniques, one of the main challenges remaining is protecting the voter. As briefly discussed in Section IV, techniques such as arithmetic encoding can be used to protect voters in TMR. We have also discussed the possibility of having redundant sandboxes monitor the results of the voter, by having read-only access to a device driver and signaling the monitor if an error is detected. We plan to compare these techniques as part of (real-time) online fault detection and recovery in Quest-V.

References

[1] A. Avizienis, "The N-version approach to fault-tolerant software," *Software Engineering, IEEE Transactions on*, no. 12, pp. 1491–1501, 1985.

[2] H. Kopetz, A. Damm, C. Koza, M. Mulazzani, W. Schwabl, C. Senft, and R. Zainlinger, "Distributed fault-tolerant real-time systems: The Mars approach," *Micro, IEEE*, vol. 9, no. 1, pp. 25–40, 1989.

[3] R. Keichafer, C. J. Walter, A. M. Finn, and P. M. Thambidurai, "The MAFT architecture for distributed fault tolerance," *Computers, IEEE Transactions on*, vol. 37, no. 4, pp. 398–404, 1988.

[4] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. Melliar-Smith, R. E. Shostak, and C. B. Weinstock, "SIFT: Design and analysis of a fault-tolerant computer for aircraft control," *Proceedings of the IEEE*, vol. 66, no. 10, pp. 1240–1255, 1978.

[5] D. Briere, C. Favre, and P. Traverse, "Electrical flight controls, from airbus a320/330/340 to future military transport aircraft: A family of fault-tolerant systems," in *The Avionics handbook*, C. R. Spitzer, Ed. CRC Press, 2001.

[6] J. Rushby, "The design and verification of secure systems," in *Eighth ACM Symposium on Operating System Principles (SOSP)*, Asilomar, CA, Dec. 1981, pp. 12–21.

[7] J. Von Neumann, "Probabilistic logics and the synthesis of reliable organisms from unreliable components," *Automata studies*, vol. 34, pp. 43–98, 1956.

[8] R. E. Lyons and W. Vanderkulk, "The use of triple-modular redundancy to improve computer reliability," *IBM Journal of Research and Development*, vol. 6, no. 2, pp. 200–209, 1962.

[9] Y. Li, R. West, and E. Missimer, "A virtualized separation kernel for mixed criticality systems," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. ACM, 2014, pp. 201–212.

[10] B. Jenkins, http://burtleburtle.net/bob/hash/doobs.html.

[11] N. Rollins, M. J. Wirthlin, P. Graham, and M. Caffrey, "Evaluating TMR techniques in the presence of single event upsets," in *Military and Aerospace Programmable Logic Devices International Conference*, 2003.

[12] P. Ulbrich, M. Hoffmann, R. Kapitza, D. Lohmann, W. Schroder-Preikschat, and R. Schmid, "Eliminating single points of failure in software-based redundancy," in *Dependable Computing Conference (EDCC), 2012 Ninth European*. IEEE, 2012, pp. 49–60.

[13] T. C. Bressoud and F. B. Schneider, "Hypervisor-based fault tolerance," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95. New York, NY, USA: ACM, 1995, pp. 1–11.

[14] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield, "Remus: high availability via asynchronous virtual machine replication," in *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*. San Francisco, 2008, pp. 161–174.

[15] M. Lu and T.-c. Chiueh, "Fast memory state synchronization for virtualization-based fault tolerance," in *Dependable Systems & Networks, 2009. DSN'09. IEEE/IFIP International Conference on*. IEEE, 2009, pp. 534–543.

[16] J. Zhu, W. Dong, Z. Jiang, X. Shi, Z. Xiao, and X. Li, "Improving the performance of hypervisor-based fault tolerance," in *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*. IEEE, 2010, pp. 1–10.

[17] F. Cristian, "Understanding fault-tolerant distributed systems," *Communications of the ACM*, vol. 34, no. 2, pp. 56–78, 1991.

[18] V. Hadzilacos and S. Toueg, "A modular approach to fault-tolerant broadcasts and related problems," 1994.

[19] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live migration of virtual machines," in *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation-Volume 2*. USENIX Association, 2005, pp. 273–286.

[20] S. K. Reinhardt and S. S. Mukherjee, *Transient Fault Detection via Simultaneous Multithreading*, ser. ISCA '00. New York, NY, USA: ACM, 2000. [Online]. Available: http://doi.acm.org/10.1145/339647.339652

[21] B. Döbel, H. Härtig, and M. Engel, "Operating system support for redundant multithreading," in *Proceedings of the tenth ACM international conference on Embedded software*. ACM, 2012, pp. 83–92.

Fast User Space Priority Switching

Alexander Zuepke, Marc Bommert, Robert Kaiser
RheinMain University of Applied Sciences, Wiesbaden, Germany
Email: {alexander.zuepke|marc.bommert|robert.kaiser}@hs-rm.de

***Abstract*—This paper presents fast user space priority switching, a mechanism for tasks to change their scheduling priority without entering the operating system kernel. Instead, tasks and the operating system kernel agree on a shared memory space storing the current task's priority. While the task changes its priority by writing to a variable in the shared memory, the operating system kernel synchronizes its internal scheduling priority with the user task's priority lazily on certain occasions affecting scheduling.**

We discuss two different protocols for fast user space priority switching. For two ARM-based platforms, we compare their implementations with a traditional approach which uses system calls to change scheduling priorities. The presented approach is suitable for systems using partitioned preemptive fixed-priority scheduling.

I. Introduction

Operating system environments, like OSEK[1] and AUTOSAR[2] in the automotive world, ARINC 653[3] in Avionics, and POSIX[4] real-time scheduling for industrial applications, often rely on preemptive fixed-priority scheduling, either on single processor systems or in a partitioned environment comprising a single processor per partition. The system's scheduled active entities, called tasks, processes, or threads, switch their scheduling priorities when entering a critical section, according to the operating system's *priority ceiling protocol* [5] to prevent deadlocks. Using precomputed priorities based on previous analysis, the tasks raise their priorities upon entering a critical section to a specific value, and lower their priorities again to the previous value when leaving the critical section. Moreover, critical sections can be acquired in a nested fashion. Typically, on occasions where priorities are changed, other important things happen as well, so most operating systems use a system call to implement the state changes in the operating system kernel. However, system calls are expensive operations on most processor architectures and in most operating systems, involving a state change of processor privileges, saving and restoring registers, or switching to different stacks.

This paper presents a concept to change task priorities in user space with low cost by avoiding system calls entirely in the common fast path. To this end, we define a shared memory protocol between user space tasks and the operating system kernel.

The rest of this paper is structured as follows: We introduce terminology, the execution environment, and the problem scenario in section II. In section III, we discuss our optimized approaches in detail. Section IV compares the approaches with a traditional implementation using system calls to change priorities on two ARM-based platforms for automotive and industrial / multimedia usage scenarios. We discuss the results in section V. Finally, section VI lists related work and section VII concludes.

[1]Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug / Vehicle Distributed Executive [1]

[2]AUTomotive Open System Architecture [2]

[3]Avionics Application Standard Software Interface [3]

[4]Portable Operating System Interface [4]

II. Problem Definition

We use the following terminology and make the following assumptions: The term *task* refers to the unit of execution scheduled by an operating system. The operating system provides two execution modes: either *user mode* for application contexts, or *kernel mode* for operating system specific activities. Additionally, the distinction of *user space* and *kernel space* denotes separated address spaces for the execution modes, e.g. user code can not access kernel code and data, and kernel code must validate user space pointers before access.

For task scheduling, we assume an execution environment using *partitioned preemptive fixed-priority scheduling*, i.e. independent scheduling on each processor. The higher a priority value, the more favoured a task is selected by the operating system scheduler. For tasks of the same priority, we assume activation in *FIFO-order* based on task arrival times.

Each task τ_i has a current *scheduling priority* $P_i(t)$ and an upper priority bound, the *maximum controlled priority* $Pmax_i$ up to which it can adjust its own priority during task execution time. For brevity, we shorten $P_i(t)$ to P_i.

We assume that a *priority ceiling protocol* is used, such that each *critical section* CS_m has a dedicated ceiling priority $Pceil_m$. This ceiling priority is statically defined as the maximum scheduling priority of all tasks that compete for that specific critical section CS_m.

We further assume that all tasks comply with the priority ceiling protocol, i.e. that no task tries to enter a critical section with a priority lower than the ceiling priority $Pceil_m$ of the section. Conversely, the maximum controlled priority of all tasks that compete for a specific critical section CS_m is higher or equal to the ceiling priority of this critical section $Pceil_m$. That is: $\forall\tau\forall m : Pceil_m \leq Pmax_i$.

As illustrated in Figure 1, on entry into a critical section at t_{enter}, task τ_i raises its priority from its current scheduling priority P_i to $P'_i := max(P_i, Pceil_m)$. On exit of that critical section at t_{leave}, it lowers its priority back to its previously possessed priority P_i. Moreover, critical sections can be entered in a nested fashion.

As Figure 1 shows, in a scenario where task τ_i raised its priority and exclusively performs resource access, while another task τ_j arrives at t_{arrive}, three possible priority relations are to be distinguished:

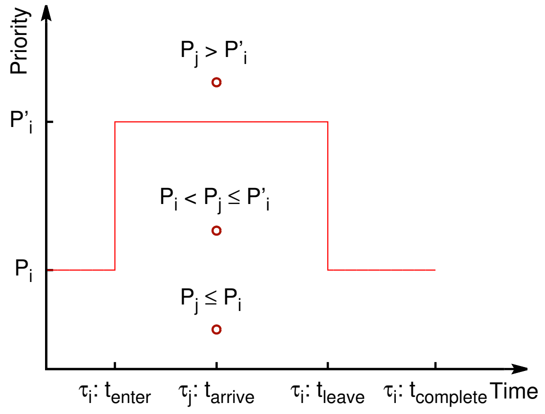


Fig. 1: Cases of interaction: τ_j arrives, while τ_i raised its priority temporarily from P_i to P'_i during a critical section.

- Case $P_j > P'_i$: The priority of τ_j is higher than the raised priority of τ_i. The operating system kernel thus immediately preempts τ_j over τ_i. τ_i will effectively be preempted while still being in the critical section. By definition of the priority ceiling protocol, all tasks with higher priority than P'_i, which includes τ_j, do not access the critical section τ_i was preempted in.

- Case $P_i < P_j \leq P'_i$: The priority of τ_j is higher than the scheduling priority P_i initially possessed by τ_i, but is lower than or equal to its raised priority P'_i. This case has to be considered by the operating system kernel since, as soon as τ_i lowers its priority back to P_i, it needs to be preempted immediately in favour of τ_j.

- Case $P_j \leq P_i$: The priority of τ_j is lower than or equal to the initial scheduling priority P_i of τ_i. Here, τ_j can not preempt τ_i regardless of τ_i's current priority. In fact, τ_j will not be selected as currently scheduled task before τ_i completes at $t_{complete}$.

In the scenario depicted in Figure 1, a traditional implementation approach would use two system calls to raise and lower a task's scheduling priority or disable interrupts during execution of the critical section. Due to the low overhead of interrupt disable/enable pairs compared to system calls, such a pragmatic approach is often considered acceptable if the execution environment permits the use of privileged instructions by application code, such as in Automotive. However, in POSIX and ARINC 653, disabling interrupts is not allowed.

To lower the cost of priority ceiling protocols, we consider the following optimization: if no task arrives while task τ_i is in a critical section, the system calls could be omitted. This could be implemented by a *lazy* approach: the step to adjust priorities is delayed to the point in time when a scheduling specific event occurs, e.g. until a new task arrives.

Two scenarios are possible: either the kernel tracks all critical sections τ_i may enter, or it tracks P_i. We consider the latter: τ_i stores its current scheduling priority in the user space variable `uprio` which is known and accessible to the operating system kernel, and the kernel synchronizes τ_i's actual scheduling priority with this value when necessary.

III. Fast User Space Priority Switching Protocols

In this section, we discuss two different protocols for fast user space priority switching, referred to as UPRIO|KPRIO and UPRIO|NPRIO, according to the names of the protocol variables.

A. *The* UPRIO|KPRIO *Protocol*

The UPRIO|KPRIO protocol comprises two variables:

- `uprio` describes the priority of the current task τ_i in user space, *user prio*, and is set by the task on priority change, i.e. when entering or leaving a critical section.
- `kprio`, shorthand for *kernel prio*, reflects the kernel's view of the scheduling priority of τ_i. On kernel entry[5], the kernel updates P_i from `uprio` and writes the resulting priority back to `kprio`.

Using this protocol, τ_i can raise its scheduling priority by issuing single memory store operation only:

```
function Protocol1_RaisePriority(prio)
    uprio := prio
```

To handle preemption correctly when lowering the priority again, τ_i needs to check (by looking at `kprio`) whether it has been interrupted since it entered the critical section and, if so, to issue a system call to trigger rescheduling in the kernel:

```
function Protocol1_LowerPriority(prio)
    uprio := prio
    if uprio < kprio
        SyscallPreempt()
```

The kernel part of the protocol synchronizes the internal scheduling priority on kernel entry:

```
function Protocol1_KernelSyncPriority()
    if uprio > currentTask.maxprio
        uprio := currentTask.maxprio
    kprio := uprio
    if kprio < nextTask.prio
        KernelPreempt()
```

For robustness, the kernel ensures that τ_i scheduling priority does not exceed the task's maximum controlled priority $Pmax_i$. Furthermore, it checks for pending task preemption.

Using this protocol, τ_i can raise and lower its scheduling priority in the fast path by issuing one memory load and two memory store operations only. In the slow path, i.e. when interrupted by the kernel in the critical section, one system call is inevitable.

In a nested critical section scenario, τ_i needs to raise its priority twice: from P_i to P'_i on entering the outer critical section CS_m, and finally to P''_i on entering the inner critical section CS_n. Using the described protocol, again no system call is involved in this step. The number of system calls on lowering the priority ranges between zero and two: zero for no interruption in the fast path, one for an interruption in the outer

[5]The kernel is entered by a processor architecture specific *trap* mechanism on system calls, asynchronous interrupts, or exceptions like division by zero.

critical section, and two for an interruption in the inner one. In the latter case, the protocol exhibits its worst case behavior: with each transition to a lower priority level, it has to issue a system call, because `kprio` > `uprio`.

B. The UPRIO|NPRIO *Protocol*

The second protocol also uses two variables, `uprio` and `nprio`, but with slightly different semantics:

- `uprio` describes the current priority of τ_i set by user code.
- `nprio`, meaning *next priority*, refers to the priority of the next eligible task for scheduling in FIFO and priority order. The kernel provides this information and updates it on every scheduling decision.

Like the first protocol, UPRIO|NPRIO allows rapid priority changes with no system calls in the fast path as long as the next thread's priority remains below P_i. If `uprio` drops below `nprio` on lowering the priority, user code issues a system call:

```
function Protocol2_RaisePriority(prio)
    uprio := prio

function Protocol2_LowerPriority(prio)
    uprio := prio
    if uprio < nprio
        SyscallPreempt()
```

The kernel updates `nprio` accordingly when tasks are inserted to or removed from the ready queue. Additionally, it bounds `uprio` to $Pmax_i$ and checks whether to preempt:

```
function Protocol2_KernelReadyQueueChange()
    ...
    nprio := nextTask.prio
    ...
    if uprio > currentTask.maxprio
        uprio := currentTask.maxprio
    if uprio < nprio
        KernelPreempt()
```

Despite the similarities, UPRIO|NPRIO only needs a system call if preemption is really required. Therefore, in the nested critical section scenario from above, the number of system calls to lower the priority is at most one.

C. Protocol Summary

When comparing UPRIO|KPRIO and UPRIO|NPRIO, both protocols use the same technique to change priorities by writing to a variable in user space, `uprio`, and checking another variable, `kprio` or `nprio`, when lowering the priority to test for (possible) preemption. While UPRIO|KPRIO uses the priority of the currently scheduled thread for that, UPRIO|NPRIO relies on the actual priority of the next eligible thread instead.

Despite being faster than an implementation using system calls to raise *and* lower priorities, implementations of both protocols add overhead to ready queue handling (synchronizing `uprio`) or context switching (setting `uprio` and `kprio`/`nprio`).

Still, UPRIO|KPRIO bears potential for optimization: instead of synchronizing `kprio` on every kernel entry, the synchronizing step could be postponed to actual scheduling decisions like in the UPRIO|NPRIO protocol.

IV. EVALUATION

We compare implementations of the approaches presented in section III to a traditional implementation using system calls to change priorities. To evaluate the implementations, we selected two ARM-based platforms from different use cases, but with a similar system architecture.

As a typical representative of an automotive processor, we selected the Hercules TMS570 evaluation board from Texas Instruments. The TMS570 has two ARM Cortex-R4 processor cores operating in lockstep mode at 180 MHz. This Cortex-R4 implementation does not have any caches, but fast, tightly-coupled on-chip SRAM for data storage and flash memory for instruction storage. Also, it supports a memory protection unit (MPU) to isolate applications. On the other end of the spectrum, the AM3358 processor on the BeagleBone Black board represents a system typically used in industrial and multimedia scenarios. Its Cortex-A8 core has split data and instruction caches of 32KB size each, a memory management unit (MMU), and it operates at 550 MHz.

Despite these differences, both processors share the same instruction set architecture and most of the exception handling model. To exclude side effects by memory management, we use a static memory layout on both processors and execute the same benchmark on a small statically configured OSEK-like operating system, based on a custom micro kernel.

We compare the two approaches UPRIO|KPRIO and UPRIO|NPRIO to each other and to an implementation using system calls for each priority change. We use two scenarios for evaluation: Firstly, in subsection A, we provide the execution times of a micro benchmark of all three approaches to determine the overhead of the fast priority switching implementation in subsection B. Secondly, in subsection C, we evaluate the benefit of fast priority switching in a nested locking scenario with and without preemption.

A. Micro Benchmarks

The micro benchmark shown in table II comprises multiple functions to evaluate the platform performance in general and

TABLE I: Processor Characteristics

Parameter	TMS570	AM3359
Typical applications	safety critical transportation applications	industrial automation, consumer electronics
CPU Core	Cortex-R4	Cortex-A8
Micro architecture	ARMv7-R	ARMv7-A
Pipeline	in order, dual-issue 8 stages, 1 ALU	in order, dual-issue 13 stages, 2 ALUs
L1 Caches	none none	2x 32 KB, 4-way 16 word line
L2 Cache	none	256 KB (not used)
Board	TI Hercules board with TMS570LS3137	BeagleBone Black rev. 1 with AM3359AZCZ
CPU Clock	180 MHz	550 MHz
SRAM Clock	180 MHz	275 MHz
Instruction fetch	Flash, 180 MHz	I-Cache, 550 MHz

basic scheduling related activities of an OSEK-like execution environment in particular. In braces, we give the relative performance gain over the traditional system call approach.

To analyze the platform performance, we conducted the following tests:

- NOP (No OPeration) loops to analyze the overhead for decrement-and-branch instructions,
- function calls followed by an immediate return, and
- memory performance of load and store operations.

To evaluate our operating system and its scheduling overhead, we measured the execution time of the following combinations of system calls:

- A *null system call* determines the overhead of system calls in general.
- `Schedule` enforces a round-trip through the scheduler without a context switch. The call puts the current task in `READY` state on the ready queue and reschedules it immediately.
- A `ChainTask` call where the calling task activates itself again: in addition to `Schedule()`, this shows the overhead of resetting the task's state.
- The `ActivateTask` / `TerminateTask` pairs shows task activation of a lower and higher priority task. Activation of the higher priority task causes scheduling and a context switch to this newly activated task, which immediately terminates, followed by another context switch back to the original caller. Activation of the lower priority task just measures the cost of placing a task on the ready queue.
- An event loop: a high priority task waits for incoming events which are signalled by a low priority task in a loop. The high priority task issues two system calls: `WaitEvent` to wait for events in the operating system kernel, and `ClearEvent` to acknowledge the events. This again comprises two context switches.

TABLE II: Micro Benchmark Results in CPU Cycles

Benchmark	System Call	UPRIO|KPRIO	UPRIO|NPRIO
TMS570			
NOP	7.5		
function call	22		
read 1K flash, 32-bit	2474		
read 1K SRAM, 32-bit	2078		
write 1K SRAM, 32-bit	2078		
null system call	157	185 (+17.8%)	157 (+0%)
`Schedule`	329	351 (+6.7%)	359 (+9.1%)
`ChainTask`	416	433 (+4.1%)	443 (+6.5%)
task activation low priority	295	320 (+8.5%)	312 (+5.8%)
task activation high priority	983	1060 (+7.8%)	1060 (+7.8%)
event loop	1166	1262 (+8.2%)	1224 (+5.0%)
AM3359			
NOP	2		
function call	7		
read 1K flash, 32-bit	786		
read 1K SRAM, 32-bit	786		
write 1K SRAM, 32-bit	786		
null system call	144	167 (+16.0%)	144 (+0%)
`Schedule`	251	283 (+12.7%)	279 (+11.2%)
`ChainTask`	295	327 (+10.8%)	324 (+9.8%)
task activation low priority	244	267 (+9.4%)	280 (+14.8%)
task activation high priority	710	776 (+9.3%)	752 (+5.9%)
event loop	917	1031 (+12.4%)	958 (+4.5%)

All tests were run in a loop 16384 times. For measurement, we used the 32-bit cycle counter of the ARM processor's performance monitor unit, which can be read in a single instruction. The cycle counter is configured to run at the processor's core clock speed.

The overall size of the operating system kernel and the benchmarks on the TMS570 board is 26 KB code and 25 KB data. The same values for the AM3359 are only slightly higher, so the overall working set fits into each of the processor's data and instruction caches.

We used the GCC 4.6.3 cross compiler for ARM provided by the Ubuntu 12.04 Linux distribution to compile our C99-based operating system. We let the compiler optimize for size with `-Os -fomit-frame-pointer` and used inline functions where possible.

B. Overhead of Fast Priority Switching Protocols

As table II shows, the Cortex-A8 core of the AM3359 possesses a faster micro architecture and shows roughly 2.5x faster memory performance than the TMS570. Also, the different protocols have no impact on platform performance. When comparing the performance of the system calls, only a performance benefit of 20% to 30% remains.

The UPRIO|KPRIO protocol shows a general overhead of up to 17.8% on the TMS and up to 16.0% on the AM3359. The UPRIO|NPRIO protocol performs better with an overhead of up to 9.1% on the TMS and up to 14.8% on the AM3359. Especially the decision to synchronize user priorities only on scheduling decisions pays off for null system calls.

The operating system kernel is not further optimized for any of the protocols. All three compared implementations use a defined region per task to host the protocol variables[6]. The shared region alone causes an overhead of 2% to 3% on context switches (values not shown).

C. Nested Locking Scenario with Preemption

To determine the overhead of critical sections and measure the effect of possible preemption on lowering a task's priority, the critical section benchmark defines three points in time when to activate another task:

T1	outside the outer critical section
T2	inside the outer, outside the inner critical section
T3	inside the inner critical section

At these trigger points, the benchmark task activates one out of four tasks whose priority value may be:

A) above the inner critical section's ceiling priority,
B) between the outer and the inner critical sections' ceiling priorities,
C) between the benchmark task's normal priority and the outer critical section's ceiling priority, or
D) lower than the benchmark task's normal priority.

Alternatively, no additional task is activated. In total, this provides 15 different procedures, of which we consider only a meaningful subset.

[6]The region between kernel and user task additionally hosts the ID of the currently executing task, which is the only value the system call variant updates

Similarly, we benchmark a non-nested critical section. We again interrupt the benchmark task outside and inside the critical section by none, another higher, medium, or lower priority task. This additionally provides 8 different procedures in total, of which we also consider only a meaningful subset. For the non-nested case, the single critical section is referred to as the outer critical section.

Table III shows the timing of nested and non-nested critical sections in terms of CPU cycles for both platforms including possible interruptions. In braces, we give the relative performance gain over the common system call approach. For both, the nested and non-nested cases, our protocols show good results when the critical sections are not interrupted, achieving a performance gain of more than 72% compared to the system call approach. Naturally, if the measured task is subject to interruptions, its execution time increases.

For interrupted critical sections, table III lists three typical combinations: *(+ highprio task)*, where a higher priority task causes immediate preemption, *(+ lowprio task)*, where a lower priority task does not cause preemption, and *(+ medprio task)* for scenarios where a medium priority task causes preemption as soon as a critical section is left. Where it makes a difference when the benchmark task is interrupted, i.e. in the outer or the inner critical section, the numbers are presented. Especially the Uprio|Kprio protocol is sensitive to this.

TABLE III: Critical Section Benchmark in CPU Cycles

Benchmark	**System Call**	**Uprio|Kprio**		**Uprio|Nprio**
		TMS570		
non-nested	370	T*	101 (-72.7%)	101 (-72.7%)
+ lowprio task	670	T1	437 (-34.8%)	424 (-36.7%)
		T2	659 (-1.6%)	
+ medprio task	1321	T1	1148 (-13.1%)	1136 (-14.0%)
		T2	1382 (+4.6%)	1299 (-1.7%)
+ highprio task	1340	T1	1157 (-13.7%)	1140 (-14.9%)
		T2	1400 (+4.5%)	
nested	747	T*	184 (-75.4%)	184 (-75.4%)
+ lowprio task	1014	T1	504 (-50.3%)	482 (-52.5%)
		T2	730 (-28.0%)	
		T3	947 (-6.6%)	
+ medprio task	1707	T1	1216 (-28.8%)	1197 (-29.9%)
		T2	1455 (-14.8%)	1367 (-19.9%)
		T3	1668 (-2.3%)	
+ highprio task	1694	T1	1223 (-27.8%)	1206 (-28.8%)
		T2	1450 (-14.4%)	
		T3	1669 (-1.5%)	
		AM3359		
non-nested	327	T*	28 (-91.4%)	28 (-91.4%)
+ lowprio task	581	T1	296 (-49.1%)	276 (-52.5%)
		T2	506 (-12.9%)	
+ medprio task	1046	T1	839 (-19.8%)	783 (-25.1%)
		T2	1040 (-0.6%)	935 (-10.6%)
+ highprio task	1061	T1	822 (-22.5%)	787 (-25.8%)
		T2	1046 (-1.4%)	
nested	651	T*	58 (-91.1%)	58 (-91.1%)
+ lowprio task	901	T1	313 (-65.3%)	293 (-67.5%)
		T2	531 (-41.1%)	
		T3	736 (-18.3%)	
+ medprio task	1376	T1	840 (-39.0%)	804 (-41.6%)
		T2	1058 (-23.1%)	957 (-30.5%)
		T3	1275 (-7.3%)	
+ highprio task	1382	T1	846 (-38.8%)	804 (-41.8%)
		T2	1076 (-22.1%)	
		T3	1275 (-7.7%)	

For Uprio|Kprio, activations inside the inner critical section (T3) are most expensive. These entail two system calls to synchronize the task's priority, whereas activations in the outer critical section (T2) just need one system call. For the TMS570, the T2-scenarios perform worse than the standard system call based approach by up to 4.6% in case of the kernel preempting the current task in favour of the interrupting task. As the number of system calls is exactly one in both the standard system call approach and our protocols, this is solely due to protocol overhead.

In contrast, the Uprio|Nprio protocol does not show these effects. As it issues at most one system call (zero for non-nested T2-scenarios) to synchronize its priority with the kernel, its results are always faster than the system call approach.

V. Discussion

In the benchmark results, both protocols show an overhead in system call performance compared to a traditional approach. This overhead needs to be justified by the performance gain of handling critical sections in user space in the fast path. We think the results draw a realistic picture of the protocols, as neither benchmarked approach was specifically optimized.

A. Benchmark Results

For Uprio|Kprio, we expected a constant overhead for each system call and only little impact on scheduling related calls. The former is an effect of synchronizing priorities on every system call, and the latter shows the overhead of updating the protocol variables. We also expected to see a *staircase* pattern due to additional system calls on lowering priorities when interrupting the protocol inside critical sections; the overheads are higher the higher the nesting level is.

However, we did not expect that Uprio|Kprio would perform so poorly and sometimes even shows slower performance than the system call based approach. We also benchmarked an implementation of the Uprio|Kprio protocol which does not synchronize priorities on every system call, but only on scheduling conditions (benchmark results not shown). The constant overhead for every system call disappears, but the worst case condition of a system call when lowering the priority still dominates the performance.

We also benchmarked the cost of updating user variables in a system call based approach to get an estimate of the costs (values not shown). We compared an implementation providing `uprio` and the ID of the current task in user accessible variables, an implementation providing just the ID, and one providing no variables. The overhead for providing the first variable (task ID) was about 4%, the additional overhead for the second variable (`uprio`) was less than 2% on a context switch, and about 4% for calls affecting the task's priority. These results are in line with the micro benchmarks.

The Uprio|Nprio protocol behaves as expected. The benchmark values show that it needs at most one system call when lowering the priority, and also the overall performance gain looks promising, especially on a system with caches. However, the benchmark results of a non-nested critical section interrupted by a medium priority task show, that the worst case timing is still in the range of the pure system call based approach. We assume this effect has prevented the adoption of such protocols in general purpose operating system.

Comparison of the results for TMS570 and AM3359 shows, that the performance gain of the protocols is higher by about 10% on the architecture with caches. We would have expected to see that the handling of the additional protocol variables would show much less impact on a CPU with caches, but the micro benchmark results point into a different direction.

At this point, it also becomes clear that further analysis of the benefits of the protocols is difficult using just these synthetic benchmarks. We need to run real-world workloads or require statistical information on the distribution of nested and non-nested locking and typical preemption patterns to see the overall effect on a long-running system.

As said before, a pragmatic approach in Automotive is to disable interrupts at the beginning of critical sections. This is probably related to higher costs of following the OSEK priority ceiling protocol compared to disabling interrupts. We need to compare the proposed protocols to these pragmatic implementations as well, as upcoming automotive platforms may no longer allow to disable interrupts in user space due to increased function safety requirements.

B. Safety and Security Considerations

From a safety point of view, the following aspects are relevant. A task τ_i can try to exceed its maximum controlled priority $Pmax_i$ by placing a higher priority value into `uprio`. The kernel must check this whenever it reads `uprio` and must bound the value to $Pmax_i$. Additionally, it is possible to enforce a lower priority bound $Pmin_i$ in an implementation, should that be a requirement. Lastly, a task can act as a foul player and not issue a system call on lowering the priority. This behavior has the same effect as a thread not leaving the critical section, because it delays the scheduling of higher priority tasks. This problem is not introduced by the fast priority switching approach: it would also happen with the traditional approach using system calls. Tasks accessing the same resource must mutually trust each other anyway.

Also, both protocols leak scheduling related information: UPRIO|KPRIO reveals that the user task has been interrupted, and UPRIO|NPRIO exposes the priority of the next eligible task for scheduling on the ready queue. This may hinder the adoption in security sensitive operation environments.

VI. Related Work

In real-time scheduling on single processor systems, *priority inversion* problems occur when a medium priority task preempts a low priority task inside a critical section, and is itself preempted by a high priority task which tries to enter that critical section. As the low priority task is not scheduled for execution, it cannot leave the critical section the highest task wishes to enter. The *Priority Inheritance Protocol* (PIP) [5] temporarily raises the priority of the lower priority task to the priority of a higher priority task when the higher priority task is waiting to enter a critical section locked by a lower priority task. The *Priority Ceiling Protocol* (PCP) [5] assigns a ceiling priority to each critical section which has to be assumed on entering the critical section. The *Stack Resource Protocol* (SRP) by Baker [6] solves this for scheduling with dynamic priorities such as Earliest Deadline First (EDF). For these protocols, multiprocessor variants exist [7] [8].

Operating system environments use these protocols or adaptions thereof. OSEK [1] and AUTOSAR [2] use the *OSEK immediate priority ceiling protocol*. POSIX [4] refers to a similar protocol as `PTHREAD_PRIO_PROTECT`.

Some commercial operating systems such as LynxOS with its "Fast Ada Page" [9] employ techniques similar to the described protocols. Previous PikeOS [10] single core implementations used a model comparable to UPRIO|KPRIO.

Linux' vDSO [11] and L4's user-level TCB [12] map a page into user space to share information such as the current thread's control block and IPC arguments (L4), or current CPU and system time (Linux).

The pending event indicator in L4's vCPU concept [13] brings fast interrupt enable / disable pairs to para-virtualized operating systems on top of a micro kernel. Its interface also uses a two-way indicator to signal the interrupt status and pending interrupts like the UPRIO|NPRIO protocol.

Sloth [14] schedules tasks as interrupts and thus delegates scheduling decisions to the interrupt controller. Tasks interface with the interrupt controller directly to change their scheduling priorities. Benchmark results show similar performance benefits for critical sections. However, this performance comes at the price: a malicious task could easily monopolize the CPU. This approach is only feasible if all tasks can trust each other.

Similarly optimized for the fast path, the *Fast User Space Mutex* (Futex) [15] supports mutexes with low overhead, requiring a system call only on contention.

VII. Conclusion

We have shown a concept enabling the currently executing task to change its scheduling priority in user space. When raising priorities, no system calls are needed. For lowering priorities again, system calls are only required when scheduling is necessary. We have described and evaluated two approaches, UPRIO|KPRIO and UPRIO|NPRIO, and discussed the safety impact in case of tasks misbehaving by exceeding their assigned priority ranges.

Both protocols show performance gains for uncontended critical sections and, in case of UPRIO|NPRIO, similar or better performance on contention.

The presented approach is suitable for real-time operating systems with partitioned preemptive fixed-priority scheduling, especially for the OSEK priority ceiling protocol in Automotive.

For future work, we would like to combine the presented approach with the Futex concept of [16] to implement fast *priority ceiling mutexes* and further synchronization primitives in user space for AUTOSAR, ARINC 653, and POSIX use cases. Also, we need to conduct measurements using more complex scenarios to evaluate the impact on real-world applications. Finally, we would like to discuss the applicability to multicore environments, mixed criticality systems, and approaches using dynamic priority scheduling such as EDF.

References

[1] OSEK/VDX, “Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug / Vehicle Distributed Executive.” [Online]. Available: http://www.osek-vdx.org/

[2] AUTOSAR, “AUTomotive Open System ARchitecture.” [Online]. Available: http://www.autosar.org/

[3] ARINC Report 653, “Avionics application software standard interface.”

[4] IEEE, “POSIX.1-2008 / IEEE Std 1003.1-2008 real-time API,” 2008.

[5] L. Sha, R. Rajkumar, and J. P. Lehoczky, “Priority Inheritance Protocols: An Approach to Real-Time Synchronization,” *IEEE Trans. Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.

[6] T. P. Baker, “Stack-based Scheduling of Realtime Processes,” *Real-Time Systems*, vol. 3, no. 1, pp. 67–99, 1991.

[7] R. Rajkumar, L. Sha, and J. P. Lehoczky, “Real-Time Synchronization Protocols for Multiprocessors,” in *IEEE Real-Time Systems Symposium*. IEEE Computer Society, 1988, pp. 259–269.

[8] P. Gai, G. Lipari, and M. Di Natale, “Minimizing Memory Utilization of Real-Time Task Sets in Single and Multi-Processor Systems-on-a-Chip,” in *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, ser. RTSS ’01, 2001, pp. 73–.

[9] LynuxWorks, “LynxOS RTOS.” [Online]. Available: http://www.lynuxworks.com/

[10] SYSGO, “PikeOS.” [Online]. Available: http://www.pikeos.com/

[11] “vDSO - overview of the virtual ELF dynamic shared object.” [Online]. Available: http://man7.org/linux/man-pages/man7/vdso.7.html

[12] J. Liedtke and H. Wenske, “Lazy process switching,” in *Proceedings of the Eighth Workshop on Hot Topics in Operating Systems*, 2001, pp. 15–18.

[13] A. Lackorzynski and A. Warg, “Virtual Processors as Kernel Interface,” in *Twelfth Real-Time Linux Workshop*, 2010.

[14] W. Hofer, D. Lohmann, F. Scheler, and W. Schröder-Preikschat, “Sloth: Threads as interrupts,” in *Proceedings of the 2009 30th IEEE Real-Time Systems Symposium*, 2009, pp. 204–213.

[15] H. Franke, R. Russell, and M. Kirkwood, “Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux,” in *Proceedings of the Ottawa Linux Symposium*, 2002, pp. 479–495.

[16] A. Zuepke, “Deterministic Fast User Space Synchronisation,” in *OSPERT Workshop*, 2013.

Implications of Multi-Core Processors on Safety-Critical Operating System Architectures

Stefan Burger, Kevin Müller, Oliver Hanka, Michael Paulitsch
Airbus Group Innovations
Munich, Germany

Andrea Bastoni, Henrik Theiling
SYSGO AG
Klein-Winternheim, Germany

Matthias Heinisch
Airbus
Buxtehude, Germany

***Abstract*—As additional hardware in airplanes increases their weight and in turn their fuel consumption, multi-core platforms are an interesting potential solution to achieve more processing capabilities onboard while avoiding carrying additional weight. However, compared to single-core platforms, multi-core platforms entail the additional price of requiring more complex components with tailored timing and communication strategies for the processes running on different cores at the same time. This paper presents the developed strategies and the lessons learnt from porting, deploying, and implementing on a multi-core platform a recent cabin management software of an actual passenger airplane and a security gateway for real-time application. As no standards and best-practices exist in the current industrial landscape, this work sets an important industrial basis for implementing and deploying safety-critical applications in multi-core environments.**

Keywords-component: Operating systems, industrial best-practice; ARINC 653; time partitioning; Avionics Software; Method Evaluation; Multi-Core Platforms

I. INTRODUCTION

There is a clear trend that future architectures of high-performance data processing and computing will be multi-core or even many-core processor platforms [1]. With increasing integration due to weight saving requirements and increasing needs of computing power due to functional integration, avionics – the electronics in aircrafts – needs to adapt to multi-core computing platforms as well. The need for functional integration of systems and the addition of new functionality (ideally without any additional weight) pushes processing performance to its limits. An example is represented by the various monitoring and support functions that have been integrated recently into many aircrafts in order to release the pilot from his routine tasks and improve airplane safety. Furthermore, customers i.e., airlines and ultimately passengers, constantly demand that more features –like high-definition in-flight entertainment (HD-IFE), Cabin Wi-Fi, etc. [21]– be available on airplanes.

However, the introduction of multi-core processing platforms into avionics is not without risks as it poses significant challenges at various levels. On single-core platforms, former issues of avionic architectures concerned system-level scheduling, communication links, and communication delays. In today's multi-core environments those issues become challenges at operating system (OS) level. We will outline this in more detail in the next section.

Nowadays, most of the costs for an airline are operational costs like fuel. Recent experience has shown that modern system design approaches can save nearly one ton of weight [2], which is a significant factor for saving fuel and hence for reducing operational costs. Multi-core CPUs have the potential to save even more Space, Weight and Power (SWaP), which are sparse in aviation platforms.

The main contributions of this paper are the investigation and evaluation of the influences from multi-core platforms into the real-time scheduling of real-world industrial applications. Such evaluations investigate multiple case studies that map real-world applications from the aerospace domain to multi-core processing platforms, and discuss the implications of possible OS-related (in detail ARINC-653-related) timing schemes and scheme changes that should be supported in the future. We present sufficient requirements and challenges for more flexible scheduling approaches in aviation systems. Given that no industrial standards and practical scheduling approaches exist so far for multi-core processors used in the safety-critical aerospace domain, this work is unprecedented and sets the basis for future development and discussions.

The analyzed case studies originate from porting an experimental cabin management and monitoring system (CMMS) and implementing a security gateway application on a multi-core platform. Such real-world applications are sufficiently challenging to evaluate different complex alternatives for timing scheduling solutions within a real-time OS. Our contributions will explain the lessons learnt and make suggestions for future multi-core avionic systems and time-partitioned OS architectures suitable for the avionics domain. Although possible scheduling solutions have been covered in literature for performance-oriented applications, the currently adopted approach in the safety-critical domain – with statically allocated execution slots as defined in the standard ARINC 653 [26] – requires a considerable configuration effort and is often not intuitive. Additionally, in order to minimize the certification costs (which correlate to complexity), real-time OS vendors adopt simple and lightweight solutions, thus further reducing many deployment scenarios that may be assumed straight forward in other domains.

The remainder of this paper has the following structure. Section II describes background information on our work. Section III gives an overview about the foundations of our evaluation and related work. Section IV explains the environment in which the evaluation has taken place. Section V illustrates our evaluation scenarios before we present the lessons learnt in Section VI. In the last section, we discuss planed future steps, and conclude our work.

II. Introduction of Multi-Core Computers to Avionics and Related Challenges

The introduction of multi-core systems in avionics causes challenges where, due to safety regulations, independent computing resources are still an important requirement. Shared resources such as memory, busses, system level caches, and chip input/output blocks – often praised as cost-saving factors in consumer electronics – can pose significant challenges in safety-driven computing cultures. A real concern is whether full *time*- and *space-partitioning* – i.e., the segregation of resources – can be guaranteed even in worst-case scenarios [3][30]. Additionally, with the reduced feature sizes of today's multi-core production processes, single event effects become more prevalent [15]. Only in recent years, the aviation certification authorities such as the Federal Aviation Administration (FAA) or the European Aviation Safety Agency (EASA) have started clarifying their positions with respect to complex electronics such as multi-core processors [4]. Nevertheless, the need for more information and experience with commercial off-the-shelf (COTS) multi-core platforms and their influence on safety-relevant functions is required in order to design hard real-time applications and to certify those systems [8]. Several projects have started gathering the required knowledge to allow future certification of multi-core platforms (e.g., RECOMP [12], EMC2 [13], ARAMiS [14][13]).

A second challenge is the migration of the currently used aviation software architectures from single-core to multi-core processors. Together with the transition from the federated architecture (*fedArch*) to Integrated Modular Avionics (*IMA*) [7], the transition to multi-core platforms can be clearly considered another challenging industrial paradigm change. In a fedArch, each system function uses its own resources i.e., each function utilizes a dedicated board (CPU, RAM, I/O etc.). IMA allows a better usage of computing resources: instead of deploying one board per function, in an IMA architecture, functions utilize a common, shared computing farm – which includes I/O – and several logical functions are encapsulated in so-called *partitions*. The most widely used OS standard for IMA development is ARINC 653 [26], defining the Avionics Application Standard Software Interface. This standard describes a layered OS environment [5] where separated partitions host different functionality with different criticality levels. These partitions are virtual containers hosting separated software. Each partition works on an individual subset of system resources such as CPU cores, I/O, and RAM. A strict, a-priori known, *static time scheduling* controls the execution-time of the partitions and allows to set hard real-time execution constraints for every application. The first airplanes using an IMA approach were the Airbus A380 and Boeing's B777. Nowadays, all modern airplanes –like the A350 or the B787– have adopted IMA system architectures in combination with ARINC 653-compliant OSs.

The next generations of airplanes will have to adapt multi-core platforms in combination with IMA and ARINC 653. However, this will force developers and system integrators to change current software architectures, in particular their communication models and scheduling processing schemes. Another challenge when using partitioned systems in combination with multi-core platforms is how to correctly devise the time scheduling approach for several partitions, the scheduling of all partitions together within one hyperperiod (*major time frame*), and the partition time frame of parallel executing partitions [10]. On a multi-core platform, partitions will be executed in parallel and therefore, in order to allow an optimized usage of resources and communication, the system designer needs to find a suitable static time-scheduling schema that fulfils the different timing requirements. Furthermore, on a multi-core platform, communication between partitions is more complex, since the software designer needs to respect the different timings in combination with applications executed in parallel. For example, he has to take into account how long a message transfer requires to reach its destination partition (possibly executing on a different core) to guarantee worst-case timing requirements.

In order to fulfill all needed safety and security requirements and therefore fulfilling the certification requirements, while at the same time containing certification costs, OSs need to reduce the configuration complexity of systems deployed on multi-core platforms. Limiting the possible timing schema options is a viable and very promising approach to enable relative straightforward configurations and to satisfy certification requirement. In section V, we present and discuss the timing schema options that would fit both the above configuration and certification requirements in the context of a recent IMA cabin management software.

III. Foundations and Related Work

This section explains all foundations and fundamentals of the evaluation system, state-of-the-art, and used techniques.

A. Native Separation-support in OSs

Given the importance of ensuring time- and space-segregated execution of partitions, it is fundamental to select an appropriate execution environment for the partitions. A microkernel-based OS meets this requirement by offering a *separation kernel* with support for scheduling of active entities, separation of memory and access control, separation of external devices, interrupt handling, and inter-thread communications. A microkernel-based OS is therefore the natural platform for the evaluation presented in this paper. Specifically, to practically discuss our approach we have chosen SYSGO's PikeOS (Version 3.2), which allows asymmetric multiprocessing (AMP) and symmetric multiprocessing (SMP) [9] in combination with ARINC 653 partitioning [6]. SYSGO's PikeOS is a commercially available microkernel based real-time OS that provides hypervisor-like virtualization capabilities and ensures strict time and resource partitioning.

Separation is achieved in PikeOS through the concepts of resource and time partitions. Resource partitions encapsulate one or more tasks,[1] which in turn encapsulate one or more threads that constitute the active scheduling entity in PikeOS [25]. A resource partition in PikeOS is a container of a set of physical resources and privileges for user applications – implemented using one or more tasks or threads. In other words a resource partition is a virtual-machine environment for

[1]A task identifies a separate address space shared by all threads assigned to it.

guest applications spacing from simple tasks to complete guest OSs. PikeOS' security and safety functionality lie in the ability of strictly separating the physical resources assigned to a resource partition.

Furthermore, each thread in PikeOS is assigned to a ***time partition***. Each resource partition can be assigned to one time partition while each time partition can be associated with several resource partitions. The relations are illustrated in Figure 1. Processing time is allocated to time partitions and threads inside one time partition share the allocated processing time, which is defined by a *static* time-window-based scheduling. [5]

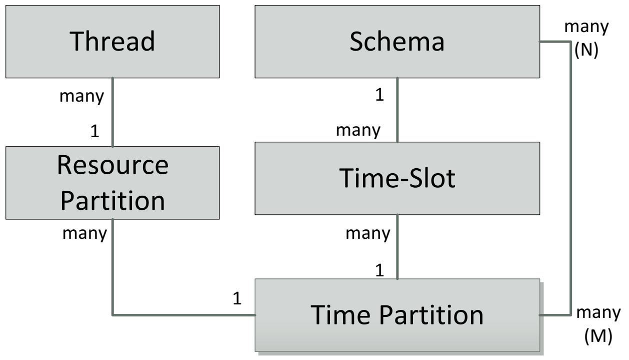


Figure 1: Relationship among resource and time partitions in PikeOS. A Time Partition can comprise multiple time slots, which form a Schema. A Schema contains therefore several Time Partitions. Threads are assigned to Resource Partitions, which in turn are assigned to Time Partitions.

An appropriate assignment of resource partitions to time partitions allows to implement of the execution model implied by the ARINC 653's major time frame concept.[2]

B. *Challenges of Scheduling a Multi-Core Systems*

Today's IMA platforms typically use single-core processors like the IBM 75x or Freescale MPC744X family. These processors have a relatively simple caching and pipeline architecture. Often, they are pure host processors requiring separate bridge chips, which are specifically developed for an individual airplane system. In contrast, most multi-core processors are COTS components with complex instruction pipelines, branch prediction, multi-level caches and cache coherency modules with included I/O and DMA controllers. These optimizations lead to better performance but lower the timing determinism.

On a single-core processor, no thread parallelism is allowed, while on a multi-core processor, parallel execution of multiple threads is the normal case. This leads to interference between applications running on different cores when hardware resources are *shared* between the cores. Resource sharing is detrimental for worst case performance: as shown in Schliecker [19] and in Fuchsen [17], the performance of one core can drop by 50% for specific applications if memory or PCI bus is excessively used by other cores. Similar results are shown in Nowotsch [3] for memory and platform-level caches.

Common multi-core processors have multi-level caches for performance optimization and they use cache coherency modules and crossbars to interconnect the cores. Software running on different cores is therefore not executing independently. Even in the absence of explicit software data or control flows between cores, resource-coupling exists at platform level due to shared hardware. A switch between resource partitions comprises a context switch on the specific cores. Caches are likely to have to be flushed and synchronized for coherency.

C. *Related Work*

Carpenter et al. [22] described challenges and complexity in theory and practice of scheduling in a safety-critical environment, which is likely to run on a multi-core processor. King [23] recently explained techniques, such as slack partitioning and cached scheduling, for the usage in safety-critical software on multi-core platforms. He argues that this is a powerful approach as applications can utilize remaining computing bandwidth in a systematic manner. Carnevali et al. [18] have presented a formal approach to design and verify two-level hierarchical scheduling systems, as used in ARINC 653. The approach includes all necessary steps from design to development of real-time systems. Schliecker et al. [19] introduced an analytical approach for calculating worst-case response times in automotive real-time system using tasks and shared resources. Nowotsch et al. 0 introduced an approach to manage multi-core Worst-Case Execution Time (WCET).

IV. Case Study Environment

In this section, we explain the two systems we used to evaluate time partitioning on multi-core platforms. Furthermore, this section introduces the used hardware and software platforms.

A. *Cabin Management System*

The cabin management and monitoring system (CMMS) of commercial passenger airplanes is the major controlling device for functionalities like cabin light, crew communication, passenger announcement, climate, water and waste etc. For the enhancement of in-flight accommodation, airlines and passengers recently demanded for more up-to-date technologies such as Cabin Wi-Fi, HD-Inflight-Entertainment Systems, or Cabin Video Monitoring. All such demanded technologies need more processing power. DO-214 [20] describes some of the minimum audio operational performance requirements of a CMMS, which rise challenging time and maximum communication delay requirements. Some of the CMMS's functions are safety-critical, i.e., the communication between the cabin server and the end devices has to be guaranteed. In order to guarantee communication in hard real-time, currently one application periodically (<100µs) sends data to the end devices. Designing a timing scheduling for a single-core platform is quite easily compared to a multi-core platform. A CMMS system has several applications with different WCETs. This generates several hundred safety requirements that must be satisfied by the system. Although on

[2] A major time frame is a repeating, fixed-length period during which each partition is executed at least once; the major time frame is therefore a multiple of the hyperperiod of all partition periods.

a single-core the time-scheduling is linear and clearly defined, it is currently a challenge to devise an optimized time-scheduling schema for new system configurations holding around 30 cabin-related applications.

The mentioned trend of deploying more cabin functionality onto one platform requires more computing performance. Current systems use single-core CPUs. Increasing the number of single-core computers to reach better performance would require more space, power consumption and weight, with undesirable impacts on the airplane. Multi-core platforms would allow minimizing size and weight in both current and future systems. Leveraging a multi-core-capable ARINC 653 OS [27], a new cabin server architecture that holds several functions integrated on one board is feasible and desired.

B. Hardware Platforms

The new experimental CMMS we implemented as a case study uses several Freescale MPC8641D CPUs with two e600 cores as evaluation platform. All CPU cards are using Ethernet for inter-CPU communication. This experimental platform implements only some important cabin functionalities, like the cabin light and crew communication, with the intention to evaluate a first version of a new ARINC 653 compliant multi-core OS.

C. Software Platform

As discussed in section III, the OS chosen for our software platform is SYSGO's PikeOS 3.2. PikeOS offers AMP and SMP support, complies with the ARINC 653 standard and is certifiable according to DO-178b [24] at the required design assurance level (DAL) (level B or C depending on CMMS system architecture and application). Furthermore, it has been already used in several safety-critical avionics devices.

V. EVALUATION SCENARIOS

In order to evaluate a multi-core test environment platform, we have designed four scenarios based on already used applications in the current CMMS. All scenarios are designed for a dual-core CPU platform, which was chosen as fundamental configuration for the next generation of experimental CMMS. Furthermore, all scenarios use four partitions. These partitions can have different WCET and hold the implementation for a CMMS use case. It is important to mention that the current application timings have been reused without considering constraints given by the current OS. This approach allowed to investigate the limits of current software and hardware, and to make suggestions for the next generation of CMMS systems. The analysis explicitly focuses on the communication overheads entailed by the different solutions, and on the trade-offs that are required due to the constraints imposed by each solution. Please note that without restricting the discussion and the presented solutions, for simplicity, multirate systems (as defined in ARINC 653) are not discussed in this paper.

A. Fixed Partition Time Frame Durations

The basic idea of the first evaluation scenario is to fix execution times (partition time frames [10]) for all partitions on both cores running at the same time. The time partitions are defined as T1 and T2, whereby the amount of execution time of T1 and T2 is variable. Our approach is using four (resource) partitions (P1, …, P4) on a dual core platform, so that every core has assigned two partitions. These numbers of applications were chosen by selecting a number of relevant CMMS applications implementing basic cabin use cases. At a given time, two partitions are executed in parallel and two partitions are idle. The parallel-running partitions are assigned to the same time partition, i.e., P1 and P3 are assigned to T1. After the end of one time partition (e.g., T1), the cores switch to the next partitions. This scenario requires that one pair of partitions ends at the same time and that the partition change must occur on both cores at the same time. illustrates the mentioned scenario. In this figure, time is progressing (cyclically) from left to right.

In this scenario, a communication within a pair (e.g., between P1 and P3 or P2 and P4) of partitions can be done without large delays. A communication between P1 and P4 needs at most the complete duration of T1 to reach P4, and an answer from P4 to P1 can require up to the duration of T2. Thus, the overall communication would need, in the worst case, T1 + T2. In this example, the major frame has a period of 250µs (T1 + T2).

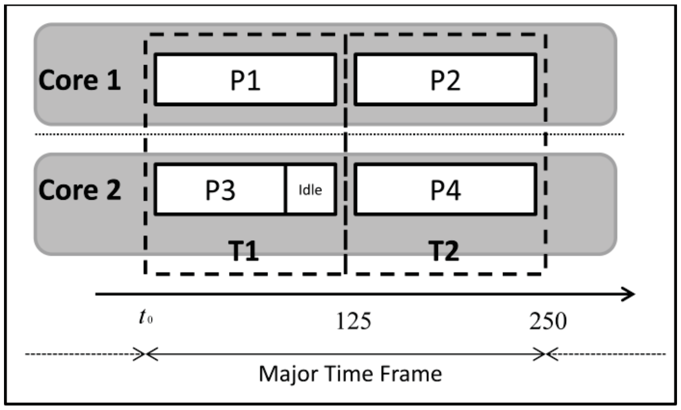


Figure 2: Schema Fix Partition Time Frames

Due to the need of switching between T1 and T2 at the same time on both cores, partition's durations has to be extended to the duration of the longest partition: if P1 has a WCET of 125 µs while P3 requires only 100µs, then P3 has to be "extended" to 125µs as well. Therefore, 25 µs of P3's execution on Core 2 will be not utilized (idle).

B. Individual Partition Time Frame Durations

This scenario uses one individual time frame for every application, which is allocated to a resource partition. All these partitions (P4, …, P7) have a different WCET. In order to allow different partition time frames on every core, every core has an independent scheduling time schema and resource partitions are assigned to independent time partitions (T3, …, T6). The different partition time frames on every core generate different major time frames: P4 (assigned to T3) on core 1 has a longer execution time than P6 (assigned to T5) on core 2. From the perspective of the system designer both cores can be seen as two independent CPUs. illustrates this scenario.

As an example, if the WCETs for T3, T4, T5, and T6 are 80µs, 50µs, 60µs, and 60µs respectively, then the major frame on core 1 is 130µs while it is 120µs on core 2. This approach influences the communication between partitions. The major time frame of core 1 is 10µs longer than the major frame on core 2. After 12 time frame periods, core 2 has executed one complete cycle more than core 1. A guaranteed direct communication between two partitions is very complex in this scenario. In a real deployment, there is the need of finding an effective tradeoff between simplifying communication and a reasonable, contained amount of idle time. A possibility would be to maintain a global major period on both cores, with different partitions executing on each core with different periodicity. In the example above, this would lead to a global major time frame of 130µs with a 10µs idle time on core 2.

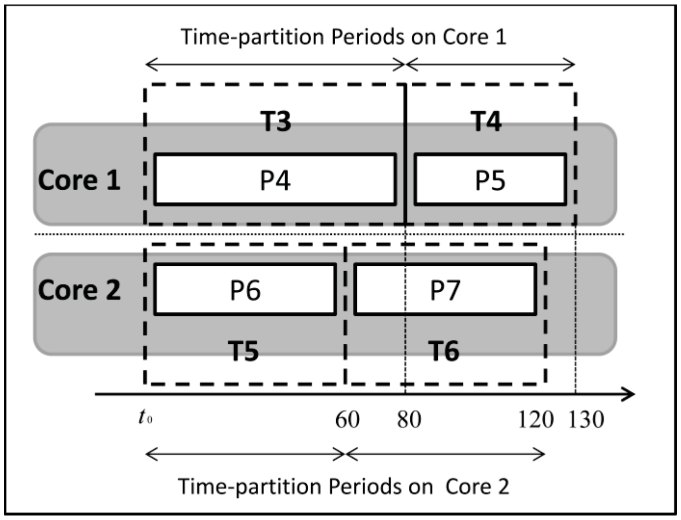

Time-partition Periods on Core 1
T3
T4
Core 1
P4
P5
Core 2
P6
P7
T5
T6
t0
60
80
120
130
Time-partition Periods on Core 2

Figure 3: Individual Partition Time Frames

C. *One Application, Two or More Cores*

The third evaluation scenario focuses on the timing of an application using more than one core. This scenario is based on the idea that an application needs more performance than those achievable on one core only. Possible examples for this scenario are image processing applications for video monitoring, communication applications for cabin phone conferences, or in general, parallelizable applications with very short execution times (e.g., around 20µs) and the requirement of a high amount of processing power. In this case, one time partition (T7) spans on all cores, and the single parallelizable application is mapped on a single resource partition (P8), allocated to both cores. The scenario is illustrated in Figure 4 (note that in the following figures, time is progressing from top to bottom).

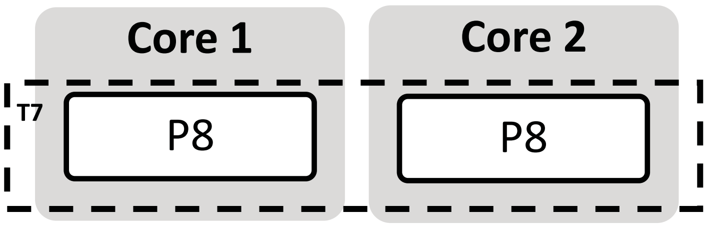


Figure 4: Schema for one Application on two Cores

From communication point of view, this scenario is straightforward: since the partition is executed on both cores, a shared memory solution is an efficient solution for the communication model. The major drawback of this solution is that it is only applicable for those applications that can be easily parallelized, while most of today's legacy applications are optimized for single-core execution. Furthermore, if dynamic elements are used in the deployment of tasks on different core, safety certification may become harder.

D. *Security Gateway Use Case*

In order to integrate more than one security domain (potentially at different criticality) in a system, and allowing the domains to securely communicate, a gateway (an additional software application) can be used to control the data flow between resource partitions in accordance with the system security policy. To realize an appropriate scheduling of such a gateway, two contradicting paradigms should be followed. While on the one hand, the gateway should execute as little as possible to avoid performance loss and allow to meet the demanding real-time requirements of the controlled applications, on the other hand, the whole system must remain deterministic and secure. Thus, the gateway needs to process the maximum amount of communication data within one scheduling period.

On single-core architectures there are two possible scheduling solutions. In the first solution, the gateway (P9) can run in its own time partition that is scheduled directly between the other communicating partitions (P10 and P11) –see . This generates a complex time-scheduling schema and may lead to some idling time overhead since the applications may not send the same amount of data in each period. The second approach is to "steal" processing time directly from the time-partition that initiated the communication. This approach adds more complexity to the WCET analysis of an application that is required by the certification process of high-criticality applications.

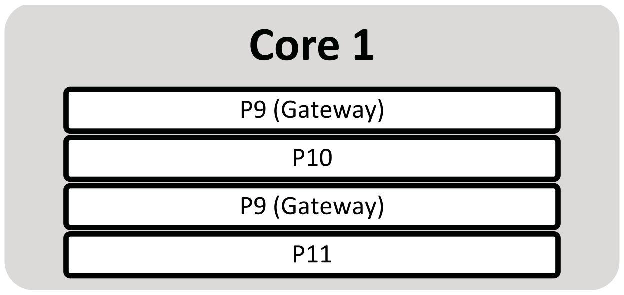


Figure 5: Single-core scheduling with Gateway processing time slot between application partitions

By using multi-core architectures it is possible to run the gateway (P9) on one dedicated core concurrently with the applications on other cores (P10 and P11) – see Figure 6. Although this approach may under-utilize the core where the gateway is allocated (as P10 and P11 do not continuously exchange messages), a higher utilization of the gateway's core can be restored by allocating lower priority tasks (or non-real-time tasks) on the same core. Thus, processing time will be employed to process the low-priority background tasks that are promptly interrupted by the gateway when a communication takes place. Although appropriate to avoid under-utilized cores, this approach could expose timing covert channel inside the system, and therefore, it requires the background low-priority tasks to be trustworthy.

A possible method to realize the described scheduling behavior within PikeOS is to use the time partition T0. In contrast to standard time partitions, threads assigned to T0 are always eligible to be scheduled concurrently with the threads belonging to the current active standard time partition. The eligible thread with the highest priority between the active standard time partition and T0 is scheduled on the core [5]. Hence, T0 can be used for event-driven real-time applications or applications without real-time requirements that can run when higher priority threads have completed. This approach is somewhat similar to the background server and slack scheduling approaches (e.g., [28], [29]) for multi-core platforms.

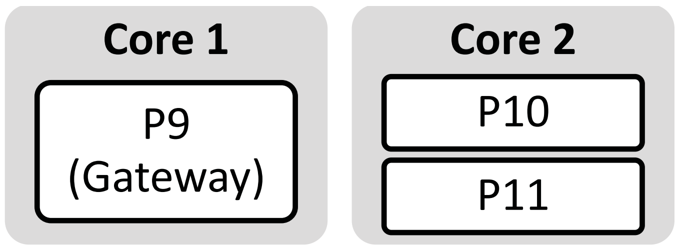


Figure 6: Multi-core scheduling with Dedicated Core for the Gateway

VI. LESSONS LEARNT

In this section, we summarize the evaluation of the scenarios mentioned in the last section. In order to evaluate the scenarios, we implemented prototypes of the CMMS applications and of the gateway application. The prototypes are sufficient to demonstrate and evaluate the usage of multi-core platforms and SYSGO's PikeOS 3.2.

A. *Fixed Time Frame Durations*

The implementation of this scenario shows that it is very challenging to find applications with the same or at least compatible WCET. All implemented applications had very different execution times and this generated high variance in the time-partitions' durations, leading to high idle times and under-utilized cores. As a "worst-case" example, during the implementation we faced the challenge of allocating on different core monitoring applications with a WCET of 250μs to be run almost concurrently with an audio streaming application with a WCET of 10ms.

An additional issue exposed by this solution is high latency entailed by concurrently flushing caches on time-partition switches on multiple cores. This is required to ensure the correct initial conditions and deterministic execution of highly safety-critical applications. Unfortunately, writing back caches into memory on all the partitions at the same point in time leads to longer write-back times due to concurrent use of the common memory and buses resources.

Another difficulty was to deal with a periodic partition that must execute after a specific amount of time. In this scenario, if a periodic application is combined with a non-periodic application, the non-periodic application must also be scheduled when the periodic partition is activated. Although this may seem a trivial task, most periodic applications have either high or low priorities. Therefore, in case of low priority periodic application, the system may waste computing time for the unnecessary non-periodic application. The opposite situation occurs in the case of high priority periodic application, as the periodic application may be executed due to the activation of the non-periodic one. In some cases, it is impossible to combine non-periodic and periodic application on the some core, because their scheduling requirements are completely incompatible.

B. *Individual Time Frame Durations*

The idea of the second time schema scenarios or variants thereof is currently not supported by any ARINC-653-compatible OS. Nonetheless, this scenario is fully flexible and would fulfill most of the needed requirements of application designers. Every core has its own scheduling schema, and time-partition assignment is independent on every core. One of the main challenges is related to the communication between applications assigned to different time-partitions on different cores. The synchronization of such communication is very challenging as each core may execute independent sequences of time-partitions. The application designer is therefore forced to closely monitor the timing of data transfer between different applications. Such task is complex and hard to configure and control as the major execution cycle of one core can be faster than the cycle(s) of the other core(s). Therefore, in this scenario, important and complex communications behaviors are more difficult to implement. Another major challenge is the current lack of precise WCET tools supporting applications running within multiple ARINC 653 partitions [16]. This makes the task of precise evaluation of the length of each independently executing time partition even harder. Improvement in this field would allow future systems to calculate the best scheduling schema by using information from applications and partitions.

C. *One Application, Two or More Cores*

As noted, the major difficulties of implementing this scenario are related to the parallelization of legacy, already certified, single-core applications. Furthermore, parallel-executing threads are not independent from each other, and may cause interference through, e.g., competing OS's system calls. Consequently, the timing of applications executing on other cores is affected. Although interference on other partitions and cores can be minimized by for example, synchronizing partition scheduling on all cores, high WCET penalties are unavoidable. It should be noted that newer versions of PikeOS (e.g., version 3.4) enable a synchronized partition switching on multiple cores, but such a feature was not available when the implementation was carried out.

D. *Security Gateway Use Case*

In case of our gateway application, we clearly see the potential benefit of using multi-core architectures for those upcoming safe and secure system designs that adds data flow controlling components. These gateway components need to execute whenever a communication between two applications belonging to differently classified security domains occurs. In order to avoid changes of currently established single-core scheduling schemas, the gateway can be processed on a dedicated core. However, since we cannot determine for all applications when communications exactly occur, the gateway is required to be executed whenever a communication request is initiated. To avoid the high utilization penalty of exclusively using a core for the gateway only, an event-driven real-time application should share the core with uncritical software (if possible and available). PikeOS' time partition T0 is a possible solution to operatively realize such a scenario and to deal with otherwise unused idle time.

T0 has been used as an always-active background time partition. Since it can contain multiple resource partitions with different priority levels, low-priority resource partitions (and therefore applications) have been assigned to T0 so that they could be scheduled during the idle time of the other time-partitions. The deterministic scheduling of other time partitions is not affected and only spare, otherwise-idle time is spent for low-priority resource partitions within T0. It should be noted that high-priority event-driven resource partitions can be assigned to T0 as well to minimize the response time after an event occurrence. In fact, such high priority partitions can preempt any lower priority partitions (in both T0 or in any standard time-partitions) at any time thus ensuring fastest possible response times. In case of this event/driven scheduling the system architect has to evaluate the potential impact on time determinism.

It should be noted that several challenges arose from the use of the T0 approach. In fact, although T0 can guarantee the execution time of a task, tasks and resource partitions assigned to T0 are preferred to equal-priority resource partitions assigned to standard time-partitions. This may lead to potential starvation of the resource partitions assigned to standard time-partitions.

VII. Future Steps

The scenarios presented in the paper are the first step in a long-term research project that aims to develop a new CMMS version for future airplane generations. One of the next steps will be the definition of new software architecture optimized for ARINC 653 and multi-core platforms. A first proposal of such architecture is presented in [11].

Furthermore, we identified some CMMS applications that could be successfully parallelized to fulfill the scenario in section V.C. Therefore, we have started to evaluate a four-core-CPUs for the experimental CMMS with limited functionality. To implement all original and new functions even more resources are needed.

Our first evaluation has successfully shown that a multi-core platform is able to hold more than the applications currently required for a CMMS. In the next steps, we would like to combine several other systems like video information and passenger Wi-Fi network in one single multi-core platform. However, this approach entails new security concerns such as those implied by combining several different security levels in one platform. In order to provide the highest security level, we will have to define new methods and techniques on the level of applications, OSs and hardware.

Strict time and space partitioning is one solution for dealing with the arising security challenges. However, these approaches can only prevent the leak of information of faulty memory accesses or timing attacks. However, since in every integrated system communication between partitions is required, the needed communication channels would be vulnerable to transfer secure data out of a security domain. An integrated gateway solution for controlling the information flow between partitions belonging to different domains similar to the one presented in scenario in section V.D is a first step towards securing such kind of communications. Finding multi-core optimized software architecture and scheduling approach for such gateways will be a challenging part of our future work.

Multiple accesses to I/O devices are additional complex activities that will require further investigation. During several tests, we discovered that the currently-in-use driver design has to be improved to support multi-core platforms and partitions. A clean driver interface is needed, which can control parallel accesses from partitions.

It should be noted that not all the presented multi-core design possibilities can be operatively realized with the currently available hardware/software combinations. On the one hand, due to the complexity of current multi-core platforms with shared hardware components it is very difficult to estimate accurate-WCET bounds that can be used to characterize the execution requirements needed by some of the identified design solutions. On the other hand, limitations of operating-systems such as the employed version of PikeOS (version 3.2, still optimized for single-core solutions) still do not offer adequate certifiable functionalities and means to efficiently implement all the proposed design solutions. However, newer and upcoming versions of OSs like PikeOS (with new interfaces and improved tools) provides (e.g., in PikeOS version 3.4) and will provide improved support for some of the major constraints identified in the presented scenarios. In particular, the ability of running individual time partitioning schemes on separate cores with the upcoming PikeOS version will allow to significantly simplify the porting and the scheduling of avionics applications to multi-cores.

VIII. Conculsion

In this paper we have presented the investigation of a multi-core platforms usage in the context of a real-world safety critical real-time environment with a strong focus on time scheduling and communication behavior. The main goal of this study was to better understand the operational difficulties and opportunities involved in finding an optimal scheduling for industrially-relevant real-time applications and to illustrate potential approaches when such applications are implemented and deployed on a multi-core platform. We have investigated four scenarios that were derived from a real CMMS taken from a state of the art Airbus plane. These scenarios can be seen as challenging archetypes for most aviation applications in a safety-critical real-time environment.

For each of the investigated scenarios, execution time and periods of the selected applications as well as communication needs between them have been discussed. Our results underline the challenges in migrating current avionics application designs from single-core to multi-core platforms while preserving appropriate time and space separation properties. For each analyzed solution, the major drawbacks have been presented.

Furthermore, we have proposed ideas on how to optimize software designs and communication models for future multi-core systems, not only in the avionics field, but in other fields with similar safety certification and performance requirements. When dealing with upcoming multi-core safety critical systems, an optimized solution for task distribution in a certifiable real-time environment will require a combination of system architecture, software architecture, operating system, and communication modules and further joint development efforts.

ACKNOWLEDGMENT

This work was funded within the projects ARAMiS by the German Federal Ministry for Education and Research (BMBF) with the funding ID 01IS11035 and SeSaM funded by the BMBF under funding ID 01BY1122 and by ARTEMIS research grant RECOMP under the research grant of the BMBF with funding ID 01IS10001.

We would like to thank our colleagues from Airbus Group Innovations in Munich, Airbus in Buxtehude, and SYSGO AG in Klein-Winternheim for their support and numerous discussions.

REFERENCES

[1] G. Lowney, Why Intel is designing multi-core processors, ACM symposium on Parallelism in algorithms and architectures, pp. 113, 2006.

[2] J.W. Ramsey. Integrated Modular Avionics: Less is More. Aviation Today, Feb. 1, 2007. Accessed on Jan. 11, 2012 at http://www.aviationtoday.com/av/commercial/Integrated-Modular-Avionics-Less-is-More_8420.html.

[3] J. Nowotsch and M. Paulitsch.Leveraging Multi-Core Computing Architectures in Avionics.9thEuropean Dependable Computing Conference (EDCC 2012). Accepted for publication.

[4] EASA, "Certification memorandum - development assurance of airborne electronic hardware," Software & Complex Electronic Hardware section, European Aviation Safety Agency, CM EASA CM - SWCEH - 001 Issue 01, 11th Aug. 2011.

[5] R.Kaiser, S. Wagner,Evolution of the PikeOS microkernel, First International Workshop on Microkernels for Embedded Systems,2007.

[6] SYSGO AG, PikeOS 3.2 Datasheet, Accessed on Feb 4, 2012 at http://www.sysgo.com/nc/products/pikeos-rtos-and-virtualization-concept/whats-new-with-pikeos-32/?cid=921&did=679&sechash=30a9f019

[7] C. Watkins, R. Walter, Transitioning from federated avionics architectures to Integrated Modular Avionics, IEEE/AIAA 26th Digital Avionics Systems Conference, 2007.

[8] P. Radojkovic, S. Girbal, A. Grasset, E. Quinones, S. Yehia, F. J. Cazorla. *On the evaluation of the impact of shared resources in multithreaded COTS processors in time-critical environments*. ACM Trans. on Architecture and Code Optimization, Jan. 2012.

[9] P.N. Leroux,R. Craig, Easing the Transition to Multi-Core Processors, Information Quarterly, Vol4, pp34-37, 2006.

[10] P. Parkinson, L. Kinnan. *Safety-critical software development for integrated modular avionics*, Embedded System Engineering, Electronic Design Automation Ltd., Vol. 11(7), pp. 40-41, 2003.

[11] S. Burger, E. Heidinger, S. Schneele, O.Hummel, M. Heinisch, W. Fischer: "Towards Higher Changeability for Airplane Cabin Software", in Proceedings of the IASTED International Conference on Software Engineering and Applications, Dallas, 2011.

[12] RECOMP Project. Reduced Certification Costs Using Trusted Multi-core Platforms. ARTEMIS Project. Project web page http://atc.ugr.es/recomp/.

[13] EMC2 Project. Embedded Multi-Core systems for Mixed-Criticality applications in dynamic and changeable real-time environments. ARTEMIS project. Project web page http://www.emc2-project.eu/.

[14] ARAMiS Project. Automotive, Railway and Avionic Multicore System. http://www.offis.de/offis_im_profil/struktur/projekte/ansicht/detail/status/aramis.html. German National Project (BMBF).

[15] C. Constantinescu. *Trends and Challenges in VLSI Circuit Reliability*. IEEE Micro Journal, Vol. 23(4), July 2003.

[16] P.J. Prisaznuk, ARINC 653 Role in Integrated Modular Avionics (IMA).. IEEE. pp. 1—E.

[17] R. Fuchsen, "How to address certification for multi-core based ima platforms: current status and potential solutions", 29th Digital Avionics Systems Conference, October 3-7, 2010.

[18] L. Carnevali,G. Lipari, A. Pinzuti, , E. Vicario,, "A formal approach to design and verification of two-level hierarchical scheduling systems", Reliable Software Technologies-Ada-Europe, 2011, Springer, vol.6652, pp.118 -131.

[19] S. Schliecker, M. Negrean, R. Ernst, "Response Time Analysis on Multicore ECUs With Shared Resources," *Industrial Informatics, IEEE Transactions on* , vol.5, no.4, pp.402-413, Nov. 2009

[20] RTCA Inc. *DO-214, Audio Systems Characteristics and Minimum Operational Performance Standards for Aircraft Audio Systems and Equipment.* Issued 3-2-1992.

[21] M. Babb, Wi-Fi in the sky, Computing & Control Engineering Journal, IET, 2004, Vol. 15, No. 2, p. 2

[22] T. Carpenter, K. Driscoll, K. Hoyme, J. Carciofini, Arinc 659 scheduling: Problem definition, Real-Time Systems Symposium, 1994., Proceedings., 1994.

[23] T. King, Slack scheduling enhances multicore performance in safety-critical applications, Electornics Design, Strategy and News (EDN), 2011.

[24] RTCA, 'DO-178B: Software Considerations in Airborne Systems and Equipment Certification', DO-178B/ED-12B, 1992.

[25] SYSGO - Embedding Innovations, PikeOS Fundamentals,Manual, 2011.

[26] Avionics Application Software Standard Interface, ARINC Specification 653, January 1997.

[27] Huyck, P., ARINC 653 and multi-core microprocessors — Considerations and potential impacts, Digital Avionics Systems Conference (DASC), 2012 IEEE/AIAA 31st , pp.6B4-1,6B4-7.

[28] S. Baruah, J. Goossens, and G. Lipari,. Implementing constant-bandwidth servers upon multiprocessor platforms. In Proceedings of the 8th IEEE Real-Time and Embedded Technology and Applications Symposium, 2002, pp. 154–163.

[29] Faggioli, D., Lipari, G., and Cucinotta, T. The multiprocessor bandwidth inheritance protocol. In Proceedings of the 22nd Euromicro Conference on Real-Time Systems, 2010, pp. 90–99.

[30] O. Kotoba, J. Nowotsch, M. Paulitsch, S. M. Petters, and H. Theiling. Multi-core in real-time systems - temporal isolation challenges due to shared resources. Proc. of Workshop on Industry-Driven Approaches for Cost-effective Certification of Safety-Critical, Mixed-Criticality Systems (at DATE Conf.), 2013.

J. Nowotsch, M. Paulitsch, D. Buehler, H. Theiling, S. Wegener and M. Schmidt. *Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement.*Proc. of 26th Euromicro Conference on Real-Time Systems (ECRTS) 2014.

Towards Hard Real-Time Control and Infotainment Applications in Automotive Platforms

Mian M. Hamayun, Alexander Spyridakis and Daniel S. Raho
{m.hamayun, a.spyridakis, s.raho}@virtualopensystems.com
Virtual Open Systems, Grenoble, France
http://www.virtualopensystems.com

***Abstract*—Recent trends in automotive systems have introduced the need for meeting both quality of service expectations and hard real-time processing guarantees. This paper addresses this trend by reviewing most popular virtualization and hard real-time solutions, and proposes promising future directions for combining both types of systems. A generic automotive use-case and its requirements are presented targeting a single hardware platform. The key virtualization solutions discussed include KVM-Preempt-RT, Para-virtualized KVM-Preempt-RT and RT-Xen. The RTOSes evaluated are RTLinux, RTAI and Xenomai, for their suitability to the automotive use-case. Finally, future directions that can bridge the hard real-time control and virtualization gap in automotive platforms are presented. Implementation and experimental evaluation of the proposed design remain as future work.**

I. Introduction

Real-time systems must respect pre-defined timing constraints and remain deterministic in nature. The validity of computations depends on functional correctness and the availability of results before pre-computed deadlines [6] [15]. A *Real-Time Operating System* (RTOS) executes real-time tasks and ensures timing constraints by strictly respecting task priorities at all times. Key features of an RTOS include interrupt latency, determinism and preemptive scheduling. A real-time system can be *unfair*, as it needs to assure certain deadlines and usually penalizes low priority tasks. Examples of real-time applications in automotive include *Electronic Stability Control* (ESC) and *Adaptive Cruise Control* (ACC).

A *General Purpose Operating System* (GPOS) such as Linux is optimized for performance in the average case rather than real-time behavior, ensuring fairness and throughput to achieve a well-balanced system for all of its users. In the automotive context, *In-Vehicle Infotainment* (IVI) encompasses a set of low priority and non-critical applications such as multimedia, games and location services, that typically require a GPOS for portability.

The contribution of this paper is a detailed survey of key virtualization and real-time technologies with the objective of combining real time functionality and multimedia applications together to meet the growing automotive market expectations. The paper is organized as follows: Section II illustrates a generic automotive use-case, with its real-time and non real-time requirements. Section III analyzes the key real-time virtualization technologies. Section IV reviews existing hard real-time solutions and their key features. Section V describes promising software architectures for future automotive platforms and Section VI concludes this paper by discussing these directions. We use the terms Security & Safety to refer to security and safety of operating system and software services, and we do not imply life-safety hard real-time systems. Moreover, the scope of discussion has been limited to automotive context, as opposed to previous surveys [13] on real-time systems.

II. RTOS Control + GPOS based IVI in Automotive

We focus on the deployment of multiple operating systems including an RTOS and a GPOS on a single hardware platform. In the automotive use-case, a car can be considered as an ”Object” playing an important role in the *Internet of Things* (IoT) arena. Moreover, high speed mobile communication in automobile platforms can enable the use of the car as a gateway for other *Connected Objects*. The latest modem technology standards such as 4G/4G+ can enable these highly connected automotive platforms. We assume that most of the *Long Term Evolution* (LTE) protocol [5] processing takes place in a dedicated hardware module, which supports high bandwidth communications ($> 450Mbps$). Applications interfacing with LTE protocol stack reside in the GPOS, thus requiring LTE module's virtualization or direct assignment to the GPOS. The RTOS takes care of *Controller Area Network* (CAN) processing, managing the high bandwidth communications on the IEEE 802.1 AVB Ethernet Bus [16].

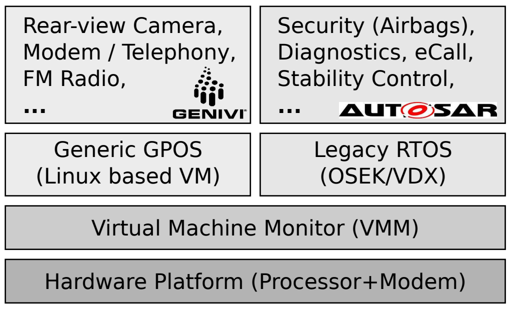


Fig. 1: GPOS and RTOS on an Automotive Platform

The use-case also assumes that RTOS and GPOS take benefit of virtualization technology and run on top of a *Virtual Machine Monitor* (VMM) similar to [19]. Stability of this use-case depends on the separation of RTOS and GPOS services, and RTOS tasks must be prioritized over the GPOS applications, in order to respect the latency and deterministic

TABLE I: Automotive Use-Case Requirements, Rationale and Explanations

Requirement	Rationale/Criteria	Explanation
Technology Type	Multi-OS Support	A Real-Time Hypervisor would seamlessly support legacy RTOS & GPOS systems.
Hardware Platform	32/64-bit Processor	Preferably 64-bit embedded processor(s) to ensure future-proof implementation.
RTOS Latency	10~50μs (approx)	Strongly dependent on the control application under consideration.
GPOS Latency	$\leq$ 5*ms* (network) + Processing	Soft Real-Time requirements mainly due to applications processing LTE packets.
Legacy RTOS Support	Easy Application Migration	Multiple legacy RTOS support or emulation to ease real-time application migration.
Disk Scheduling	Low Latency Disk Accesses	Multimedia application access times to disk resources should be minimum.
I/O Virtualization	Host/Guest Coordinated I/O	Virtualization of I/O resources should benefit from Host/Guest coordination.
Certification	Real-time OS Protection	RTOS tasks should be isolated from IVI applications to ensure certifiable operation.
Open Source	Community Building/Maintenance	Open Source software is better maintained and serves larger user base.

requirements. Figure 1 shows the generic architecture of this use-case and Table I gives some of its key requirements.

III. Virtualization Solutions with Real-time

Most of these solutions modify the Linux kernel to introduce real-time capabilities including preemption, lower interrupt latency, faster context switching and fine-grain time management. The most popular solution in this category is the Real-Time Preemption patch [1] (*a.k.a.* Preempt-RT), and others including MontaVista Linux [20] and TimeSys Linux [18]. The key idea is to enable preemption in all possible kernel contexts, instead of a few preemption points in order to build a highly responsive system. Examples of kernel blocking contexts include critical sections and *Interrupt Service Routines* (ISRs). These solutions cannot provide hard guarantees on the timing latencies of the patched kernels [23], as analysis of the whole kernel code paths is infeasible. Other real-time virtualization solutions include baremetal hypervisors [24] with a strong focus on real-time scheduling of virtual machines.

A. KVM with Preempt-RT

The Preempt-RT patch [1] takes a standard Linux kernel and modifies it to enable *almost* full kernel preemption (reentrancy). The key changes introduced are: the use of mutexes instead of spinlocks, threaded interrupts and support for *Priority Inheritance Protocol* (PIP) to avoid priority inversion. Mutexes guarding the critical sections support priority inheritance mechanism, thus they can be preempted and put on sleep mode. Interrupt handling in kernel thread contexts enables their scheduling as normal Linux processes, thus ISR preemption is supported. On average, the interrupt latency ranges between 20μs and 30μs, with a maximum value from 62μs to 336μs [8][17]. Additional Preempt-RT features include High Resolution (HR) Timers, Preemptible Soft IRQs and Preemptible RCU (Read Copy Update) mechanism.

Most of the Preempt-RT features have already been integrated into the mainline kernel [12] except for the bit spinlocks [3] and threaded interrupts that are still maintained separately. The Preempt-RT solution is implemented in a generic and portable way thus supporting a new platform does not require platform specific modifications, if the target platform is already supported by Linux. Nevertheless, some of device drivers and kernel sources may need to be reviewed and adapted.

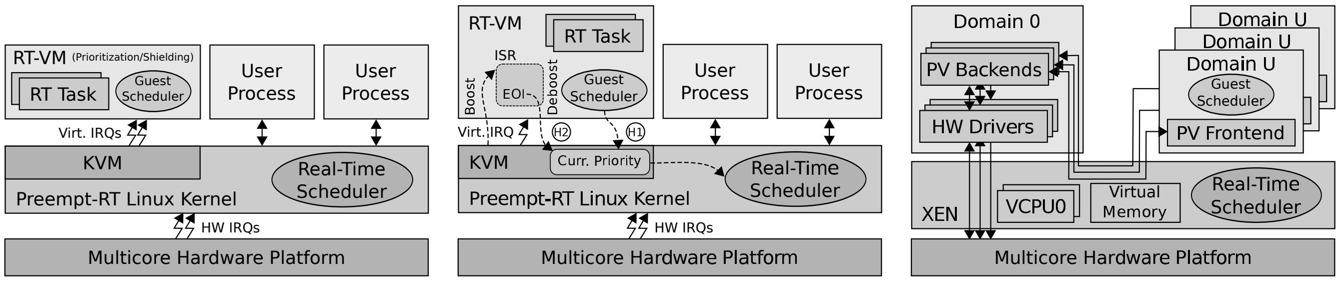
Preempt-RT patch can be employed to make the Host Linux an RTOS, and use KVM virtualization for meeting the non real-time application requirements [12]. This solution however does not *easily* support a legacy RTOS, as the real-time domain resides on the Host side rather than the Guest. To overcome this limitation, we can exploit KVM's virtualization model as discussed in [28]. Thus, a careful *Prioritization* of VMs' priorities and scheduling class can give real-time behavior in Guests. However, priorities alone cannot guarantee the real-time behavior in Guests, as interrupts on the Host machine can even preempt highest priority tasks, thus Preempt-RT patch is necessary to convert Host ISRs into kernel threads.

On SMP or multi-core platforms, CPU *Shielding* can be used to dedicate a Host CPU to a Guest RTOS VM, using the task and interrupt affinities. The RTOS VM is assigned to a CPU except CPU0, in order to avoid the common Host interrupts such as the ones originating from disk and network devices. Additionally, SMI, ACPI and tracing features of the Host kernel are disabled to avoid power management induced performance degradation. Swapping of Guest memory pages is also disabled. Figure 2(a) shows the generic architecture of Preempt-RT based virtualization solutions.

B. Para-virtualized KVM with Preempt-RT

Prioritization of virtual machines into the real-time scheduling class is one way to improve the real-time responsiveness of Guest virtual machines. Prioritization requires careful balancing and deactivation of certain features on Host machine, as real-time Guests may depend on non real-time Host services, such as kernel events and *Asynchronous I/O* (AIO). Additionally, livelocks may occur if an in-appropriate maximum priority is selected. For example, a High Resolution timer thread needs to have higher priority than the real-time virtual machine, if the VM wishes to receive events from the HR timer thread [14]. Likewise, we need to ensure that *only* real-time tasks are executed inside the real-time VMs, as 'low priority' background jobs within a real-time VM take priority over the seemingly high priority Host processes.

These issues have been analyzed in [14], which proposes a para-virtualization scheduling interface to overcome the priority inversion problems. This solution introduces two new hypercalls that real-time Guests invoke to inform the hypervisor about their internal state. The first hypercall *i.e. Set Scheduling Parameters* is used to inform the hypervisor of the current Guest process priority and policy. The hypervisor



(a) RT-KVM: Preempt-RT Host kernel (b) RT-KVM: Preempt-RT with Para-virtualization (c) RT-Xen: Baremetal VMM with RT Scheduler

Fig. 2: Virtualization Solutions with Real-Time Features

uses this information to calculate the Host-level priority of the corresponding VCPU thread and informs the scheduler about it. The second hypercall is used from Guest interrupt handling context. Before injecting an interrupt into a Guest, the hypervisor *boosts* the corresponding VCPU's priority to maximum, and the Guest informs hypervisor about the end of interrupt handling by using the *Interrupt Done* hypercall. The hypervisor then restores the previous priority and policy for this particular VCPU, known as *deboost*. Figure 2(b) shows the overall architecture of this solution.

Although, this solution avoids priority inversion but it comes with additional cost as the average latency increases to $86\mu s$ with a maximum of $434\mu s$ [14]. Moreover, the KVM para-virtual MMU operations have to be disabled as their implementation is incompatible with changes in the Preempt-RT MMU subsystem. Lastly, this solution cannot support unmodified legacy RTOSes, as the Guest OS sources must be modified for para-virtual interface implementation.

C. *RT-Xen: A Real-Time Baremetal Hypervisor*

Xen [7] is a popular open source baremetal hypervisor and allows for the co-existence of multiple domains (VMs) on the same hardware platform. It creates a special domain known as *Domain 0* that acts like a Host OS and manages Guest domains *i.e. Domain Us* in Figure 2(c). Xen uses para-virtualization so Guest OS and I/O drivers are modified to send requests to the VMM instead of hardware. The VMM forwards these requests to the para-virtual backend drivers in *Domain 0*, which then uses real hardware drivers for completing I/O operations.

Real-time virtualization in Xen *i.e.* RT-Xen [24] has been proposed for single-core CPUs and a more recent work *i.e.* RT-Xen 2.0 [25] for multi-core platforms. In all cases, the reported results show *soft* real-time behavior in virtual machines and a scheduling quantum of $1ms$ is proposed. Results in [24] show a 40% deadline miss rate when the scheduling quantum is set to $10\mu s$ and Guest systems cannot even boot for smaller scheduling intervals. A recent study [10] has reported interrupt latency (for a short 8 seconds test) of around $14\mu s$ with a maximum value between $21.2\mu s$ and $41.5\mu s$, depending on the type of scheduler used in RT-Xen. Lastly, certification of RT-Xen is still an open issue, which is an important requirement for the automotive use-case.

IV. Real-Time Solutions without Virtualization

This section provides a detailed review of key real-time solutions that are interesting *w.r.t.* the automotive use-case. In all of these solutions, a small "nano-kernel" is introduced underneath the usual Linux kernel and focuses on the hard real-time requirements of the system.

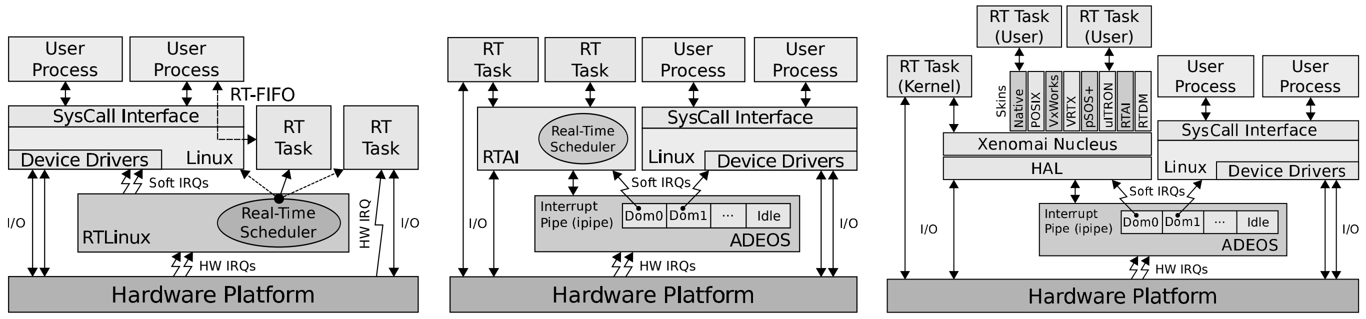
A. *RTLinux: The First Real-Time Linux*

RTLinux is the first solution to employ the dual kernel approach [27], *i.e.* the Linux kernel is treated as a low priority process and all real-time activity is confined to the RTLinux kernel space [21]. A later effort introduced POSIX threads API into RTLinux to enable user mode real-time threads. RTLinux is available in two different flavors *i.e.* the RTLinux/GPL and Real-Time Core. The overall architecture of RTLinux/GPL is shown in Figure 3(a).

The RTLinux micro-kernel is fully preemptible and uses fixed priority scheduling with support for *Priority Ceiling Protocol* (PCP) to prevent priority inversion. Linux kernel is assigned the lowest priority and executes as an idle task. The real-time scheduler supports Round Robin scheduling for tasks with the same priority. Moreover, the RT scheduler can be replaced dynamically by a more suitable one, to meet the needs of a given application.

RTLinux handles all hardware interrupts necessary for deterministic processing and propagates other ones to the Linux kernel, if interrupts are enabled by Linux. To take hardware control, the Linux kernel is modified to remove hardware management instructions *e.g.* interrupt enable and disable instructions. These instruction are replaced with hooks to render control to the RTLinux micro-kernel, which takes appropriate action *e.g.* enabling or disabling interrupt delivery to the Linux kernel. RTLinux uses the concept of *Hardware Abstraction Layer* (HAL) and provides an API for managing hardware interrupts and writing ISRs. The average latency ranges between $5\mu s$ to $10\mu s$, with a maximum latency of $27\mu s$ [21].

The major drawback of RTLinux is its proprietary design and closed source nature [21] (at-least the RTCore). The real-time tasks execute outside the Linux kernel and use a specific real-time kernel API. As a result, the GNU/Linux programming model cannot be preserved when porting real-time applications from another RTOS to RTLinux. This solution requires custom device drivers in the real-time domain and does not



(a) RTLinux/GPL Architecture (b) RTAI/ADEOS Architecture (c) Xenomai/ADEOS Architecture

Fig. 3: Hard Real-Time Solutions using a Micro-kernel

support MMU, so virtual memory and memory protection are unavailable. Inter-process communication between RTLinux tasks and Linux processes takes place through message queues (RT-FIFOs) or shared memory.

B. RTAI: Real-Time Application Interface

RTAI [2][9] follows the dual kernel approach of RTLinux and introduces a *Real-Time Hardware Abstraction Layer* (RTHAL) for simulating the hardware platform and making the Linux kernel believe that it runs on a real platform. The RTHAL gathers all pointers to the time critical kernel data and functions, such as the hardware interrupt flags, interrupt vectors, *etc.*, into a single data structure and modifies the Linux kernel routines to initialize the RTHAL pointers. RTAI then modifies the Linux kernel to use RTHAL for all of its hardware-specific operations, forwarding control to RTAI in all such cases.

More recent versions of RTAI are based on the *Adaptive Domain Environment for Operating Systems* (ADEOS) [26]. The ADEOS nano-kernel schedules multiple instances of the same or different operating systems running on top of it, and allows for the co-existence of multiple prioritized domains. ADEOS implements a pipeline scheme, which is used to virtualize the data and functions previously gathered by the RTHAL layer. Each domain, for example RTAI and Linux, is represented by an entry in the ADEOS interrupt pipeline (I-pipe), and every event entering the pipeline is delivered to all of the registered domains according to their respective priorities. RTAI resides in the highest priority domain to ensure real-time behavior. The average latency for the original RTAI solution is around $5\mu s$ with a maximum latency of $10\mu s$. Using the ADEOS layer, the average latency is increased to $10\mu s$. Figure 3(b) shows the generic architecture of RTAI using the ADEOS layer.

The ADEOS layer pushes the Linux kernel out of the Ring 0 to Ring 1 on x86 machines, by modifying the GDT entries. This aspect can cause problems in case of virtualization, especially on systems with only two levels of privileges *i.e.* system and user, as the Host is usually assumed to have full control of the hardware platform.

C. Xenomai: The RTOS Chameleon

The Xenomai [11][21] framework is yet another micro-kernel based real-time, abstract and architecture neutral operating system. Xenomai uses a generic and reusable emulation layer known as *nucleus*. The nucleus exports an architecture independent interface for controlling the hardware resources to lower-level HAL layer, which implements it in a host-specific manner. The nucleus includes necessary support for *Inter Process Communication* (IPC), and communication with Linux domains takes place through message pipes. Additionally, the nucleus provides dynamic memory allocation support with real-time guarantees and any number of memory heaps can be maintained dynamically.

Xenomai also uses the ADEOS layer for interrupt virtualization and managing different domains; receiving interrupts from hardware platform and forwarding them to higher layers, according to their respective priorities. The average latency for Xenomai on x86 machines varies between $10\mu s$ to $15\mu s$, whereas the worst case latency is reported to be $37\mu s$ and $90\mu s$ [8] in kernel and user modes, respectively.

Xenomai introduces the concept of *Skins* or personalities and each skin emulates the system call interface of a particular RTOS. The objective of a skin is to ease the portability of real-time applications from another RTOS to Xenomai framework, with minimal or no changes. Xenomai provides emulation skins for Native, VxWorks, pSOS+ *etc.* and supports many architectures including ARM[1], Blackfin and i386. The overall design of Xenomai framework is shown in Figure 3(c).

The real-time task scheduler of Xenomai uses fixed priorities with preemption support and employs PIP to prevent priority inversion. The task rescheduling procedures are locked during an ISR execution, in order to preserve atomicity of interrupt handling, and ISRs always take priority over tasks. Automatic task migration between Xenomai and Linux is possible, but requires compatibility between the used APIs *i.e.* as a function of the used skin.

Xenomai does not support MMU [21], thus virtual memory and protection is unavailable for real-time tasks. Xenomai patched kernels show improved performance due to the deactivation of power management and frequency scaling functions [17]. To improve platform support, Xenomai is aiming towards integration with the Linux Preempt-RT patches [21].

[1]Limited support for ARM architecture as of today.

[2]These latency values are for x86 machines, as reported in literature.

[3]FSMLabs and its key product *i.e.* RTLinux was sold to Wind River in February 2007 [4] but got discontinued since August 2011.

TABLE II: Comparison of Virtualization and Real-Time Technologies

Criteria	KVM/RT	PV-KVM/RT	RT-Xen	RTLinux/RTCore	RTAI/ADEOS	Xenomai/ADEOS
Technology Type	Hosted VMM Prioritization	Hosted VMM Para-virtualization	Baremetal VMM Para-virtualization	Dual Kernel (Linux + RTLinux)	Dual Kernel (Linux + RTAI)	Dual Kernel (Linux + Xenomai)
Supported Platforms	All (Potential Porting Issues)	All (Potential Porting Issues)	ARM, ARM64, i386, x86_64	Alpha, ARM[1], i386, MIPS, PPC, Proprietary	ARM[1], Cris, i386, m68k, MIPS, PPC, x86_64	ARM[1], Blackfin, i386, SH, PPC32/64, NIOS2, x86_64
Avg. Latency[2]	20~30μs	~86μs	~14μs	5~10μs	5~10μs	7~15μs
Max. Latency[2]	62~336μs	~434μs	21~42μs	10~27μs	10~20μs	10~37μs
API Support	Standard User/Kernel APIs	Standard User/Kernel APIs	Standard User/Kernel APIs	POSIX Kernel APIs	RTAI Kernel & LXRT User APIs	Multiple Common APIs User/Kernel Mode (Skins)
Open Source/Cost	Yes / Free	Yes / Free	Yes / Free	No / Licensing Fee	Yes / Free	Yes / Free
Develop. Status	Requires Patching	Requires Patching	Relatively New	Stable	Stable	Stable (Relatively New)
Support/Type	Good/Forums	Limited/Forums	Limited/Forums	Good/Vendor	Limited/Forums	Good/Forums
Deployment/ Maintenance Cost	High (Patched Host kernel only)	High (Patched Host/Guest kernel)	High (Patched Host/Guest kernel)	Low~Medium (Proprietary kernel)	Medium~High (In-house kernel)	High (In-house kernel)
Legacy RTOS/ RT Applications	Yes (No changes to Guest kernels)	No (Guest kernel require changes)	No (Guest kernel require changes)	No (RT tasks require porting)	No (RT tasks require porting)	Yes (RT Tasks only, using Emulation Skins)
Source Compatibility	Linux 3.12 Standard Drivers	Linux 3.12 Standard Drivers	RT-Xen 2.0/Xen 4.3, Linux 3.9, PV FE/BE	Discontinued[3] Proprietary Drivers	RTAI 4.0, Linux 3.8 Custom Drivers	Xenomai 2.6.3, Linux 3.10 Custom Drivers

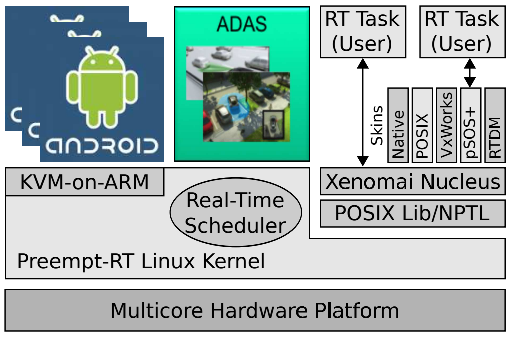
This solution, commonly known as Xenomai/forge, is currently under development and does not require the ADEOS layer as it is based on a single kernel approach. Table II compares all of the virtualization and real-time techniques discussed in Sections III and IV.

V. Architectural Directions for Real-Time and Virtualization in Automotive Platforms

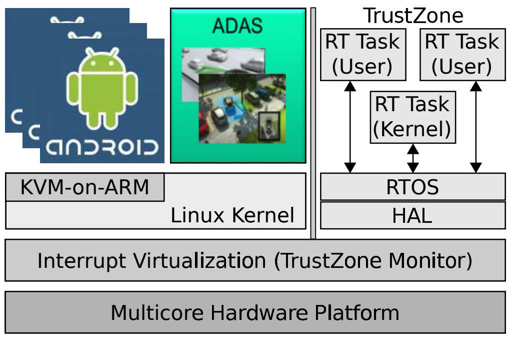
The ARMv8 architecture targets both high-end and low energy embedded platforms and provides a 64-bit computing environment while keeping backward compatibility for 32-bit systems. It provides a dedicated hypervisor mode with hardware extensions for virtualization and *Large Physical Address Extension* (LPAE) support for enabling very large address spaces. It also includes a separate execution environment known as TrustZone that can be used for meeting secure computing requirements. These features make ARMv8 an ideal candidate for virtualization as well as dual kernel solutions. In this section, we discuss promising future directions, as shown in Figure 4, that will satisfy the use-case requirements presented in Section II. Some other applications such as *Advanced Driver Assistance Systems* (ADAS) can also directly execute on the Host system, if POSIX compatible, without requiring any real-time or virtualization support.

A. Preempt-RT Host with Embedded RTOS Kernel

One possible solution is to follow the Xenomai/forge direction, where the RTOS kernel is embedded in the Preempt-RT Host, as shown in Figure 4(a). The main advantages of this solution will include support for multiple RTOS APIs, real-time virtualization, its open source nature and MMU-based address-space separation/protection for real-time tasks. On the down side, it will be (most likely) a soft real-time solution, as dependencies will exist between RTOS and Host systems. Additionally, maintenance costs will be high, as every new revision of Linux kernel will have to be reviewed for Preempt-RT patches. Lastly, certification is completely infeasible as a sound *Worst Case Execution Time* (WCET) analysis cannot be performed for such a complex system.



(a) GPOS on KVM & RTOS on RT kernel



(b) GPOS on KVM & RTOS in TrustZone

Fig. 4: Potential Architectures for the Co-existence of RTOS and GPOS on ARMv8 Platforms

B. RTOS in TrustZone with Vanilla Host

To ensure hard real-time requirements, the solution proposed in Figure 4(b) is more appropriate as it will keep the virtualization and real-time domains separate. The RTOS will execute in TrustZone on top of an Interrupt Virtualization layer similar to ADEOS, while keeping the possibility to share memory with GPOS in the non-secure world. The key advantages

TABLE III: Comparison of Proposed Architectural Directions in Figures 4(a) & 4(b)

Requirement	Proposition in Figure 4(a)	Proposition in Figure 4(b)	Favored Solution	Rationale/Explanation
Technology Type	Single RT kernel with Hosted VMM	Dual kernel with Hosted VMM	A	Lower maintenance cost
Hardware Platform	ARMv7/v8	ARMv7/v8	A & B	–
RTOS Latency	Expected $< 500\mu s$ (Soft)	Expected $< 50\mu s$ (Hard)	B	Hard real-time
GPOS Latency	Expected $< 5ms$ (Soft)	Expected $< 5ms$ (Soft)	A & B	–
Legacy RTOS Support	Yes (Skins for RT Applications)	Yes (With/Without HAL)	B	No emulation required
Disk Scheduling	Host/Guest Coordinated Scheduling	Host/Guest/RTOS Coordination	A	Reduced complexity
I/O Virtualization	Virtualization/Direct Assignment	Virtualization/Direct Assignment	A & B	–
Certification	Completely Infeasible	Feasible	B	Security & Isolation
Open Source	Yes	Yes (Partially)	A	Potentially closed RTOS components

of this solution include hard real-time guarantees, support for legacy RTOS, hardware based security, lower maintenance cost and feasibility of certification for real-time software. However, the virtualization layer will remain non real-time and execute on top of the unmodified Host kernel. In order to meet the soft real-time requirements of LTE packet processing application on GPOS, a mechanism similar to coordinated scheduling [22] between RTOS and VMM based GPOS will be investigated. Table III compares the two proposed solutions.

VI. Conclusion and Future Directions

We conclude this paper, by highlighting the advantages of solution proposed in Section V-B. Although this solution will cost more *w.r.t.* initial development effort, it is better suited to the hard real-time requirements of our automotive use-case. Furthermore, thanks to its segregated nature it will require lower maintenance effort. Our next objective is to address the development of this solution, based on the first ARMv8 SoC platform, meanwhile continuing our proof-of-concept implementation on the available ARMv7 SoCs.

Acknowledgment

This research work has been supported by the Seventh Framework Programme (FP7/2007-2013) of the European Community under the grant agreement no 610640 for the DREAMS project.

References

[1] Preempt-RT Patch on The Linux Kernel Archives. [Online] https://www.kernel.org/pub/linux/kernel/projects/rt/.

[2] RTAI Homepage, Real-Time Application Interface. [Online] http://www.rtai.org/.

[3] The future of realtime Linux. [Online] http://lwn.net/Articles/572740/.

[4] Wind River Acquires Hard Real-Time Linux Technology from FSM-Labs. [Online] http://www.windriver.com/news/press/pr.html?ID=4261.

[5] Long Term Evolution Protocol Overview, 2008. [Online] http://www.freescale.com/files/wireless_comm/doc/white_paper/LTEPTCLOVWWP.pdf.

[6] J. Altenberg. Using the Realtime Preemption Patch on ARM CPUs. In *11th Real-Time Linux Workshop (RTLW '09)*, pages 229–236, Sep 2009. Dresden, Germany.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, pages 164–177, New York, NY, USA, 2003.

[8] J. H. Brown and B. Martin. How fast is fast enough? Choosing between Xenomai and Linux for real-time applications. In *12th Real-Time Linux Workshop (RTLW '10)*, Oct 2010. Nairobi, Kenya.

[9] L. Dozio and P. Mantegazza. Linux Real Time Application Interface (RTAI) in low cost high performance motion control. In *Associazione Nazionale Italiana per l'Automazione (ANIPLA '03) (National Italian Association for Automation)*, Mar 2003. Milano, Italy.

[10] H. Fayyad-Kazan, L. Perneel, and M. Timmerman. Evaluating the Performance and Behaviour of RT-Xen. *International Journal of Embedded Systems and Applications (IJESA)*, 3(3):19–34, Sep 2013.

[11] P. Gerum. Xenomai - Implementing a RTOS emulation framework on GNU/Linux. Technical report, Apr 2004. [Online] http://www.gna.org/projects/xenomai/.

[12] S. Ghosh and P. Sampath. Evaluation of Embedded Virtualization on Real-time Linux for Industrial Control System. In *13th Real-Time Linux Workshop (RTLW '11)*, pages 173–180, Oct 2011. Czech Technical University, Prague.

[13] Z. Gu and Q. Zhao. A State-of-the-Art Survey on Real-Time Issues in Embedded Systems Virtualization. *Journal of Software Engineering and Applications*, pages 277–290, Apr 2012.

[14] J. Kiszka. Towards Linux as a Real-Time Hypervisor. In *11th Real-Time Linux Workshop (RTLW '09)*, pages 205–214, Sep 2009. Dresden, Germany.

[15] K. Koolwal. Myths and Realities of Real-Time Linux Software Systems. In *11th Real-Time Linux Workshop (RTLW '09)*, pages 13–18, Sep 2009. Dresden, Germany.

[16] R. Kreifeldt. AVnu Alliance White Paper: AVB for Automotive Use, 20 Jul 2009. [Online] http://www.avnu.org.

[17] N. Litayem and S. B. Saoud. Impact of the Linux Real-time Enhancements on the System Performances for Multi-core Intel Architectures. *International Journal of Computer Applications*, 17(3):17–23, March 2011. Published by Foundation of Computer Science.

[18] D. Locke. A TimeSys Perspective on the Linux Preemptible Kernel, Version 1.0, 2002. [Online] http://www.stillhq.com/pdfdb/000471/data.pdf.

[19] H. Mitake, Y. Kinebuchi, A. Courbot, and T. Nakajima. Coexisting Real-time OS and General Purpose OS on an Embedded Virtualization Layer for a Multicore Processor. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC '11)*, pages 629–630, 2011.

[20] MontaVista. Native Real-Time Linux Kernel Platform. [Online] http://www.mvista.com/solution-real-time.html.

[21] T. Paz. Evaluation of Xenomai and RTLinux, INT-409-TSP-0001 rev 1.0. Technical report, Magdalena Ridge Observatory, New Mexico Tech, 801 Leroy Place Socorro, NM 87801, USA, 13 Dec 2006.

[22] A. Spyridakis and D. Raho. On Application Responsiveness and Storage Latency in Virtualized Environments. In *Fifth International Conference on Cloud Computing, GRIDs, and Virtualization (CLOUD COMPUTING 2014)*, Venice, Italy, May 2014.

[23] N. Vun, H. F. Hor, and J. W. Chao. Real-Time Enhancements for Embedded Linux. In *14th IEEE International Conference on Parallel and Distributed Systems (ICPADS '08)*, pages 737–740, 2008.

[24] S. Xi, J. Wilson, C. Lu, and C. Gill. RT-Xen: Towards real-time hypervisor scheduling in Xen. In *Proceedings of the International Conference on Embedded Software (EMSOFT '11)*, pages 39–48, Oct 2011.

[25] S. Xi, M. Xu, C. Lu, L. T. Phan, C. Gill, O. Sokolsky, and I. Lee. Real-Time Multi-Core Virtual Machine Scheduling in Xen. Technical report, Washington University, WUCSE-2013-109, October 2013.

[26] K. Yaghmour. Adaptive Domain Environment for Operating Systems. [Online] www.opersys.com/ftp/pub/Adeos/adeos.pdf.

[27] V. Yodaiken. The RTLinux Manifesto. In *Proceedings of The 5th Linux Expo, Raleigh, NC*, Mar. 1999.

[28] J. Zhang, K. Chen, B. Zuo, R. Ma, Y. Dong, and H. Guan. Performance Analysis Towards a KVM-Based Embedded Real-Time Virtualization Architecture. In *5th International Conference on Computer Sciences and Convergence Information Technology*, pages 421–426, 2010.

Mixed-Criticality on Multicore (MC^2): A Status Report

Namhoon Kim, Jeremy P. Erickson, and James H. Anderson
Department of Computer Science, The University of North Carolina at Chapel Hill
{*namhoonk, jerickso, anderson*}*@cs.unc.edu*

***Abstract*—The MC^2 (mixed-criticality on multicore) framework has been proposed and implemented in LITMUS$^{\mathrm{RT}}$, a real-time extension to Linux. The implemented MC^2 framework has been used in several research efforts pertaining to multiprocessor real-time systems. This paper describes the current status of work on MC^2. There are currently two MC^2 branches. We describe the features of each branch and report on current progress in unifying these branches.**

I. Introduction

Future embedded real-time systems are expected to require increased computational workloads and functionalities. Multicore platforms have the potential to meet these requirements by offering greater computational capabilities and advantages in size, weight, and power (SWaP). However, the introduction of multiple processing cores makes real-time resource allocation more difficult. To further complicate matters, many avionic and automotive embedded applications require tasks to be supported at different criticality levels, such as safety critical, mission critical, and best effort, on a single multicore system [1].

Because the failure of a safety critical task may cause a fatal failure of a system, such a task may be provisioned with a very pessimistic worst-case execution time (WCET). This can result in wasted computational capacity due to the difference between the predicted WCET and the actual execution time observed at run time. A technique to minimize this discrepancy has been proposed by Vestal [2]. He proposed the *multi-criticality (or mixed-criticality) task model*, which provides varying degrees of WCET assurance. Specifically, for low-criticality tasks, he proposed using less pessimistic WCETs for schedulability analysis, while for high-criticality tasks, he proposed using more pessimistic WCETs.

In the RTCA DO-178B and DO-178C software standards for avionics, criticality levels range from A (highest) to E (lowest) and are determined for a system component (e.g., a task) by examining the effects of failures. Mixed-criticality scheduling on multicore platforms was first considered by Anderson et al. [3]. They proposed operating-system (OS) infrastructure that allows mixed-criticality applications to be supported on a multicore platform, assuming the five criticality levels of DO 178B/C, while ensuring real-time correctness. In follow-up work, researchers at UNC Chapel Hill and Northrop Grumman Corp. proposed a mixed-criticality scheduling framework for multicore platforms, called MC^2 (mixed-criticality on multicore), and provided schedulability analysis results [4]. In MC^2, higher-criticality tasks are viewed as "slack generators" that use only a small fraction of their execution budget. Lower-criticality tasks execute using this slack. MC^2 also employs a two-level hierarchical scheduling approach, in which *containers* (also called *servers*) [3] are used to enable the temporal correctness of subsystems.

The first implementation of MC^2 was described by Herman et al. [1], who discussed design tradeoffs and evaluated the robustness of the implemented mixed-criticality scheduler with respect to breaches in execution-time assumptions. MC^2 is implemented within LITMUS$^{\mathrm{RT}}$, a real-time extension of Linux that was designed to support real-time workloads on multicore platforms [5], [6], [13], [14].

In order to make safety-critical cyber-physical embedded systems more predictable, cache-management techniques were proposed by Ward et al. [7]. Specifically, they proposed two cache-management techniques, called *cache locking* and *cache scheduling*, and showed that the usage of such techniques can reduce WCETs in higher-criticality tasks. Ward et al. developed a branch of MC^2 in which these cache-management techniques are used, and presented experimental results on a multicore Tegra3 ARM machine.

As mentioned by Burns and Davis [8], a task may exceed its predicted level-l WCET. When such guarantees are violated, overload can occur. To provide guarantees in such overload situations, a recovery mechanism was proposed by Erickson et al. [9] that uses virtual time. This mechanism has been incorporated in a branch of MC^2 that employs a virtual timer [4]. This branch was used to obtain experimental results on an x86 machine. However, due to the different microarchitectures of MC^2 with cache management and with virtual time, these two branches of MC^2 have not yet been unified.

In this paper, we report on the current status of MC^2. In the rest of this paper, we provide relevant background (Sec. II), describe the current two branches of MC^2 (Sec. III), and then conclude (Sec. IV).

II. Background

In this section, we provide necessary background on multiprocessor real-time scheduling, LITMUS$^{\mathrm{RT}}$, and the MC^2 architecture. We also briefly explain the container abstraction used for hierarchical scheduling and common MC^2 features.

A. Multiprocessor Real-time Scheduling

Task model. We assume that temporal constraints for tasks can be modeled by the implicit-deadline *periodic* or *sporadic task model*. Under the periodic task model, a system is comprised of a set of recurring tasks. Each such task τ_i releases a succession of *jobs*, denoted $\tau_{i,0}, \tau_{i,1}, \ldots$, and is defined by a *period*, p_i, and an *execution time*, e_i. Successive jobs of τ_i are released every p_i time units, starting at time 0, and a job released at time *t* must complete by its *deadline*, $t + p_i$. Under the sporadic task model, each task τ_i is specified by an execution cost, e_i, a *minimum separation* between successive job releases, p_i, and a *relative deadline*, d_i. Task τ_i's *utilization* is given by $u_i = e_i/p_i$. We sometimes assume a *harmonic task system* wherein all task periods are integer multiples of the smallest task period.

We assume a hardware platform with *m* processors. A task system is *schedulable* on such a platform under a given scheduling algorithm if no deadline constraint is violated. In a *hard real-time* (HRT) system, jobs must never miss their deadlines, while in a *soft real-time* (SRT) system, some deadline misses are tolerable. If a job $\tau_{i,j}$ released at $r_{i,j}$ completes execution at time *t*, then its *response time* is $t - r_{i,j}$ and its *tardiness* is $max\{0, t - d_{i,j}\}$. In the definition of SRT assumed in MC^2, tardiness is required to be bounded.

Partitioned and global scheduling. Under partitioned scheduling, tasks are statically assigned to processors and migration is not allowed, while under global scheduling, tasks may migrate across processors. Generally, partitioned scheduling is preferable in HRT systems, and global scheduling is preferable in SRT systems [10], [11]. Partitioned approaches have lower run-time overheads, but processing capacity may be wasted due to bin-packing problems. In contrast, global approaches eliminate bin-packing issues and are particularly effective in SRT systems where some deadline misses are allowed [12]. A drawback of global scheduling is increased OS overheads associated with contention of shared scheduler state.

B. LITMUS$^{\mathrm{RT}}$

MC^2 is implemented in LITMUS$^{\mathrm{RT}}$, an extension to the Linux kernel that supports real-time schedulers as plug-in components [13], [14]. LITMUS$^{\mathrm{RT}}$ was developed as an experimental platform for research on multiprocessor real-time scheduling and synchronization. Time-based events, such as job releases, are handled by Linux's high resolution timer application programming interface (hrtimer API) and scheduling events and synchronization requests are handled by plug-in event handlers. LITMUS$^{\mathrm{RT}}$ provides a very light-weight event tracing tool called *feather-trace* to record scheduling events and synchronization requests [15]. The partitioned *earliest-deadline-first* (P-EDF) and global EDF (G-EDF) schedulers have been implemented in LITMUS$^{\mathrm{RT}}$ previously [16]. As noted earlier, there are currently two branches of MC^2 implemented in LITMUS$^{\mathrm{RT}}$.

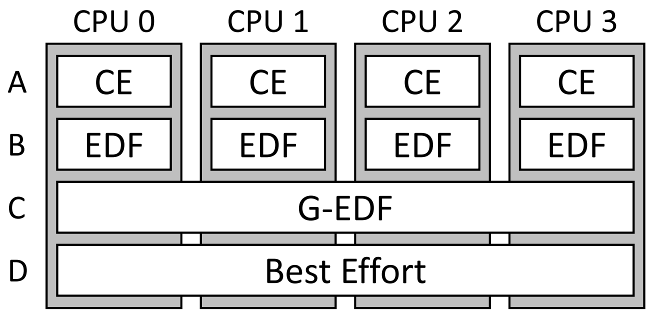


Fig. 1. Container allocation and the scheduler for each container under MC^2 on a four-processor system.

C. MC^2 Architecture

Vestal proposed a technique for eliminating under-utilization of processors due to very pessimistic WCET values [2]. He observed that the WCET values for higher-criticality tasks are needlessly pessimistic from the perspective of checking the schedulability of lower-criticality tasks. He proposed specifying per-criticality-level WCET values for each task. That is, each task τ_i has an execution time, $e_{i.l}$, for each criticality level *l* (depending on the scheduling scheme, it may not be necessary to specify an execution time for a task at higher criticality levels than its own). The level-*l* utilization of τ_i is defined as $u_{i.l} = e_{i.l}/p_i$. This has come to be known as the mixed-criticality task model. In the variant of MC^2 described here, four criticality levels are assumed, denoted A through D. MC^2 was designed with avionics workloads in mind and these workloads tend to be harmonic in nature [1].

Container abstraction. An essential part of mixed-criticality scheduling is that lower-criticality tasks should not affect higher-criticality tasks. This is very related to the concept of *temporal isolation*. Such isolation can be achieved by supporting a container (or server) abstraction within the OS. In mixed-criticality scheduling, a container is a group of tasks that is a isolated from the rest of the system [3]. MC^2 uses a two-level hierarchical scheduling approach. When the scheduler selects the next task to run on a processor, it first selects the highest-priority container among the containers that may execute tasks on that processor. Then, the scheduler selects the highest-priority task from the selected container, according to the associated scheduling algorithm of the container. The assumed containers and their associated scheduling algorithms are illustrated in Fig. 1, and explained below.

In MC^2 as proposed by Herman et al. [1], tasks are assumed to be periodic. Each level-*l* task τ_i is implemented as a single-task container within a container for its level. A task τ_i is assigned a *budget* equal to its execution time for its own level. The budget is consumed when the associated task executes and is replenished at time 0 and every p_i time units. Budget enforcement is enabled by default, but it can be disabled.

Level A. Level-A tasks are the highest-priority tasks in MC^2. They are statically assigned to processors and scheduled by a predefined dispatching table similar to the *cyclic executive* scheduling approach [17]. There are m level-A containers, one per processor. The schedulability analysis of level A is straightforward. Because level A is statically prioritized over all other levels, its schedulability is not affected by any other containers and is guaranteed at run time unless a level-A task τ_i exceeds it level-A budget, $e_{i.A}$.

If there are no level-A tasks to run on a processor at a given instant, then MC^2 considers level-B tasks. If a level-A task completes before its assigned level-A budget has been exhausted, then MC^2 allows a lower-criticality task to run for the duration of the remaining budget. This technique is known as *slack shifting* [1]. The completed job whose remaining budget is being consumed by a lower-criticality task becomes a *ghost job*. The ghost job completes when its remaining budget is equal to 0.

Level B. Similarly to level A, each processor has a level-B container. Level-B tasks are scheduled in EDF order. Optionally, *rate monotonic* (RM) scheduling can be used at level B. When no higher-criticality tasks are eligible to run on a processor, or when a level-A task is running as a ghost job, the scheduler selects the next job to run from the level-B container if such a job is available on that processor. It is required that the period of all level-B tasks is an integer multiple of the level-A *hyperperiod* (the least common multiple of level-A task periods) [4].

Level-B schedulability is achieved when the total level-B utilization of level-A and -B tasks on each processor is at most 1, since level-B scheduling across the system resembles the P-EDF scheduler and has similar theoretical properties. Level-B schedulability is guaranteed at run time unless some level-A or -B task exceeds its level-B execution time. Similarly to level-A jobs, a level-B job becomes a ghost job when it completes before exhausting its level-B budget; once it is a ghost job, its budget can be consumed by lower-priority tasks.

Level C. Level-C tasks are globally scheduled by the G-EDF algorithm. There is one global level-C container to which all level-C tasks are assigned. The G-EDF scheduler can be invoked on any processor whenever level-A or -B tasks are not eligible to run on that processor.

A level-C schedulability test is given in [4] assuming level-C execution times. Level-C schedulability is guaranteed at run time as long as no level-A, -B, or -C task exceeds it level-C execution time. Like higher-criticality levels, slack shifting is employed at level-C to allow level-D tasks to run.

Level D. Level-D tasks are scheduled on a best-effort basis. Such tasks are normal Linux tasks, which are not considered to be HRT or SRT tasks. Thus, there is no container for level-D tasks and no schedulability test is provided for this level. Level-D tasks can be scheduled by a stock Linux scheduler when there are no eligible real-time tasks to run.

Interrupt master. Dedicated interrupt handling, where all interrupts are directed to a designated processor called the *interrupt master* [16], can improve schedulability [1]. If an interrupt master is used, all release and timer events occur on the interrupt master. This enables budgeting for level-A and -B tasks on the other processors to be less pessimistic, but level-A and -B tasks on the interrupt master suffer from interrupt handing overheads. MC^2 supports using an interrupt master as an optional feature.

Timer merging. In a harmonic task system, multiple jobs are released frequently at the same time because the period of all tasks are integer multiples of the smallest period in the system. LITMUS$^{\mathrm{RT}}$ [5] uses a timer to release real-time jobs. In Linux, it is not guaranteed that all local timers start at the same time. Due to this local-timer error, tasks at levels B and C may have the same release time, yet their release timers may fire in reverse-criticality order. In this case, a level-C task is scheduled to run and then a level-B task is released and scheduled to run, which preempts the previously scheduled level-C task. To avoid this unnecessary preemption, an optional feature called *timer merging* was proposed by Herman et al. [1]. If enabled, release events within 1 μs of one another are merged using an $O(1)$ hash table operation. However, a global lock is required to merge all timer events across multiple processors. Thus, this feature can be enabled only when the interrupt master is enabled, which redirects all release events to a single processor.

Fine-grained locking. Each level-B and -C container has its own release queue and ready queue, and each level-A container has an associated dispatching table. Moreover, each processor has state indicating the currently scheduled task. The scheduler state data in MC^2 must be synchronized across processors to support MC^2's hierarchical scheduling approach. To access this state, spin locks are used to synchronize data structures on a per-container and per-processor basis [1]. The `rt_domain_t` data structure in LITMUS$^{\mathrm{RT}}$ is used to implement the ready and release queues needed to support containers. Fig. 2 provides an illustration.

If we do not carefully optimize synchronization, then MC^2 might suffer from significant overhead since the described state is accessed frequently. To mitigate this overhead, Herman et al. proposed a fine-grained state-locking mechanism [1]. This mechanism ensures two properties: (a) a processor lock can never be held for more than $O(1)$ time; and (b) container locks are never nested inside other container locks. Details are provided in [1].

III. Current MC^2 Branches

In this section, we discuss the two current branches of MC^2 implemented in LITMUS$^{\mathrm{RT}}$.

A. MC^2 with Virtual Time

In this subsection, we describe a branch of MC^2 in which virtual time is supported [9]. This version has been im-

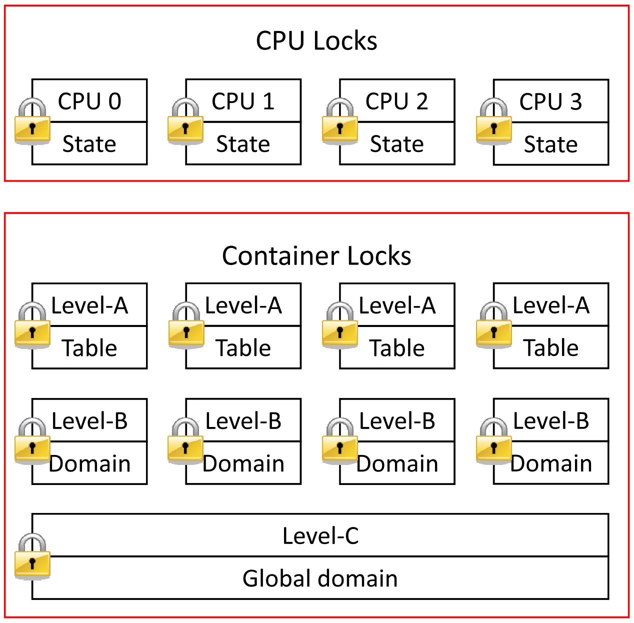


Fig. 2. Spin locks for containers and processors in the MC^2.

plemented based on the LITMUS$^{\mathrm{RT}}$ 2011.1 release (Linux 2.36.4). The usage of virtual time provides a mechanism to recover from overload of the level-C subsystem (described in more detail below), which can occur when any job at or above level C overruns its level-C WCET. This branch assumes a sporadic task model for level-C tasks since changes to the rate of virtual time alter the job-release frequencies at level C. In this branch, the interrupt master and timer merging are not supported.

Virtual time. Erickson et al. [9] modified MC^2 to support recovering from overload at level C. This can occur when any job at or above level-C overruns its level-C execution time.[1] In this situation, all successive job response times might increase permanently. Such ill effects can be dealt with by changing scheduling decisions until the overload situation has abated. In this MC^2 branch, such decisions are altered by using the idea of *virtual time* from Zhang [18] and Stoica et al. [19], where job releases are determined by a virtual clock that can change speeds with respect to the actual clock. Virtual time $v(t)$ is based on a global speed function $s(t)$. When a task overruns its level-C execution time and its response time exceeds a given tolerance value, the level-C scheduler slows down virtual time and reduces the job-release frequency at level C. The MC^2 branch with virtual time is comprised of a kernel component that manages virtual time and a userspace component that monitors job releases and completions. The kernel component controls job releases based on virtual time. The userspace component, called a *monitor* program, collects job-release and -completion information from the kernel to detect an overload situation or an idle instant. The monitor program is responsible for determining the virtual-clock speed. This virtual-clock mechanism affects only level-C tasks, not level-A or -B tasks. Detected idle instants are used to determine when recovery is completed, at which point virtual-time speed returns to actual-time speed. Experimental results show that the scheduling overheads and the execution time of the userspace monitor program are small. The introduction of virtual time increases scheduling time by about 40% on average and by 100% in the worst case. Each invocation of the monitor program completes in approximately 1 ms in the worst case [9].

[1]If budget enforcement of level-C tasks is disabled, a level-C job can overrun its level-C WCET. Even with budget enforcement, level-A and -B tasks can overrun their level-C WCETs.

GEL-v scheduling. In the MC^2 variant proposed by Mollison et al. [4], level-C tasks are scheduled by using G-EDF. As noted by Erickson et al. [20], other *G-EDF-like (GEL)* schedulers can provide better response-time bounds, so in this branch, arbitrary GEL schedulers are allowed at level C. Furthermore, since the virtual clock is used to manage job releases of level-C tasks, a modified version of GEL scheduling, called *GEL with virtual time* (GEL-v) scheduling, and a generalized version of the sporadic task model, called the *sporadic with virtual time and overload (SVO) model*, are used for level-C tasks. Under GEL-v scheduling, each job $\tau_{i,k}$ is prioritized on the basis of a virtual *priority point* (PP), and each task τ_i is characterized by a minimum separation time $T_i > 0$, and a relative PP $Y_i \geq 0$, both with respect to virtual time. At time 0, $s(t)$ is equal to 1, which means that actual time and virtual time progress at the same rate. However, when an overload is detected, the scheduler decreases $s(t)$, which reduces the progress of virtual time. As explained above, this slows down the rate of future job releases of level-C tasks, and creates extra slack to enable the system return to normal behavior.

B. MC^2 with Cache Management

Another MC^2 branch with cache management has been implemented by Ward et al. [7]. In this branch, several shared-cache management techniques have been implemented assuming a quad-core ARM machine. However, this MC^2 branch only supports the level-B and -C subsystems. The MC^2 branch with cache management uses *cache lockdown* mechanisms that requires hardware support. This is why this MC^2 branch works only for a specific ARM platform, which provides the needed cache lockdown instructions.

Cache management. Ward et al. [7] proposed a cache management technique that preallocates the dynamic memory a job uses before the job begins execution. *Page coloring* is used to allocate the memory pages required by a job. Under page coloring, a color is assigned to each page to control the mapping address of the page. The pages that have different colors map to different cache sets, so they cannot conflict with each other in the last level cache (LLC). This is used in conjunction with cache lockdown to prevent active pages for being evicted during the execution of the job. The cache is treated as a shared resource that can be either preemptive or non-preemptive, yielding two possible cache allocation policies: *cache locking* and *cache scheduling*. Under cache locking, the processors and the cache are not preemptible, while under cache scheduling, they are. In this MC^2 branch, cache management techniques are applied to level-B tasks. These cache-management techniques are not applied at level C, which is provisioned using less pessimistic WCETs. The MC^2 scheduler loads the memory pages of the next level-B job to execute into the shared LLC and flushes the pages used by the previous job. This results in additional scheduling overheads. However, it has been shown that cache management enables significant schedulability gains by reducing level-B WCETs.

Resource sharing. Cache locking ensures that the pages required by a job reside in the cache during the entire duration of its execution. This policy requires a multiprocessor real-time locking protocol: the cache is treated as a shared resource that has k replicas as given by the number of cache ways. The RNLP [21], which optimally supports the simultaneous locking of replicated resources, is used for this purpose. For example, if a job requires r pages with the same color, then it must lock r replicas of that color. Also, the job may require several colors simultaneously. To support these requirements, this MC^2 branch uses *dynamic group locking* as proposed by Ward et al. [22] in the context of the RNLP. Dynamic group locks allow a job to lock multiple resources with one lock request rather than requesting each resource individually in a nested fashion, which can increase system-call overhead and blocking times. The RNLP controls all colors and ways by using a FIFO queue for each way of each color. The maximum duration of blocking for all cache colors is $O(mr/k)$ where k is the number of ways available and r is the maximum number of ways per color requested by any job [21].

C. MC^2 in LITMUS$^{\mathrm{RT}}$ 2014.1

The previous two branches of MC^2 in LITMUS$^{\mathrm{RT}}$ have not been merged because they require different microarchitectures and the MC^2 patches are based on different Linux kernel versions (2.6.36 and 3.0.0). We are currently trying to unify the two branches with their features as a kernel configuration. This work is not finished at this time. We discuss some of the issues in unifying both branches in this section.

Container implementation. MC^2 ensures temporal isolation by supporting a container abstraction. However, LITMUS$^{\mathrm{RT}}$ currently does not support such an abstraction. The previous MC^2 implementation considers a real-time task as a container. Thus, the data structure `rt_param` in LITMUS$^{\mathrm{RT}}$ has extra variables to support container functions, such as replenishment and consumption. The MC^2 branch with virtual time uses `real_release` and `real_deadline` variables to keep track of a job's release and completion time, while the MC^2 branch with cache management uses another `rt_job` data structure, `user_job`. We need to merge these two different data types to support the container abstraction. This approach fulfills its requirements, but it is hard to trace the behaviors of both a container and each individual task in the container, and this implementation is not well-suited to job handling in LITMUS$^{\mathrm{RT}}$.

More fine-grained locking. As shown in Fig. 2, there is a lock for each domain. The domain structure at levels B and C includes release and ready queues. The scheduler is required to hold the ready-queue lock when a task is added to the release queue and vice versa. This domain locking at levels B and C should be more fine-grained. The locks at level A are fine-grained enough because the domain structure at level-A only has a dispatching table.

Cyclic executive scheduling table. Making a scheduling table for level-A tasks is quite complicated now. We currently use the Linux *proc* file system to construct a table, and we must change the LITMUS$^{\mathrm{RT}}$ scheduler plugin several times whenever changing a budget or adding a new task. We want to devise a more convenient way to manipulate the scheduling table. In both MC^2 branches, the scheduling table can be accessed by `read_proc_t` and `write_proc_t` function pointers. However, in Linux 3.10 (the base version of LITMUS$^{\mathrm{RT}}$ 2014.1), the structure `proc_dir_entry` does not have those function pointers anymore. We need to implement `proc_fops` operations to unify the two branches.

IV. Conclusion

In this paper, we have discussed the current status of work on MC^2, the first mixed-criticality scheduling framework implemented on multicore platforms. Due to the different microarchitectures and base kernel versions in LITMUS$^{\mathrm{RT}}$, two branches of MC^2 exist. We hope that the unified MC^2 we are constructing will provide more features and portability as a mixed-criticality research testbed.

In future work, we plan to extend MC^2 to allow tasks to acquire locks and have precedence constraints, in order to enable more realistic workloads. In addition, we hope to ease or remove the hardware dependency of the cache-management MC^2 branch. The cache-management MC^2 branch requires cache-lockdown instructions, which are not widely supported. We plan to investigate cache allocation mechanisms to remove the preloading and flushing of memory pages whenever a job is scheduled.

References

[1] J. Herman, C. Kenna, M. Mollison, J. Anderson, and D. Johnson. RTOS support for multicore mixed-criticality systems. In *Proceedings of the 18th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 197–208, April 2012.

[2] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *Proceedings of the 28th IEEE Real-Time Systems Symposium*, pages 239–243, December 2007.

[3] J. Anderson, S. Baruah, and B. Brandenburg. Multicore operating-system support for mixed criticality. In *Proceedings of the Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, April 2009.

[4] M. Mollison, J. Erickson, J. Anderson, S. Baruah, and J. Scoredos. Mixed-criticality real-time scheduling for multicore systems. In *Proceedings of the 7th IEEE International Conference on Embedded Software and Systems*, pages 1864–1871, June 2010.

[5] LITMUS$^{\mathrm{RT}}$ homepage. http://www.litmus-rt.org/.

[6] B. Brandenburg. *Scheduling and Locking in Multiprocessor Real-Time Operating Systems*. PhD thesis, The University of North Carolina, Chapel Hill, 2011.

[7] B. Ward, J. Herman, C. Kenna, and J. Anderson. Making shared caches more predictable on multicore platforms. In *Proceedings of the 25th Euromicro Conference on Real-Time Systems*, pages 157–167, July 2013.

[8] A. Burns and R. Davis. Mixed criticality systems - a review. http://www-users.cs.york.ac.uk/~burns/review.pdf, December 2013.

[9] J. Erickson, N. Kim, and J. Anderson. Recovering from overload in multicore mixed-criticality systems. In submission, 2014.

[10] B. Brandenburg, J. Calandrino, and J. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, pages 157–169, December 2008.

[11] H. Leontyev and J. Anderson. Generalized tardiness bounds for global multiprocessor scheduling. *The Journal of Real-Time Systems*, 44(1):26–71, February 2010.

[12] U. Devi and J. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. *The Journal of Real-Time Systems*, 38(2):133–189, February 2008.

[13] B. Brandenburg, A. Block, J. Calandrino, U. Devi, H. Leontyev, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A status report. In *Proceedings of the 9th Real-Time Linux Workshop*, pages 107–123, November 2007.

[14] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS$^{\mathrm{RT}}$: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proceedings of the 27th IEEE Real-Time Systems Symposium*, pages 111–123, December 2006.

[15] B. Brandenburg and J. Anderson. Feather-trace: A light-weight event tracing toolkit. In *Proceedings of the 3rd International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, pages 20–27, July 2007.

[16] B. Brandenburg and J. Anderson. On the implementation of global real-time schedulers. In *Proceedings of the 30th IEEE Real-Time Systems Symposium*, pages 214–224, December 2009.

[17] T. Baker and A. Shaw. The cyclic executive model and ADA. *The Journal of Real-Time Systems*, 1(1):7–25, 1989.

[18] L. Zhang. Virtual clock: A new traffic control algorithm for packet switching. In *Proceedings of the ACM Symposium on Communications Architectures & Protocols*, pages 19–29, September 1990.

[19] I. Stoica, H. abdel-Wahab, K. Jeffay, S. Baruah, J. Gehrke, and C. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, pages 288–299, December 1996.

[20] J. Erickson, J. Anderson, and B. Ward. Fair lateness scheduling: Reducing maximum lateness in G-EDF-like scheduling. *The Journal of Real-Time Systems*, 50(1):5–47, 2014.

[21] B. Ward and J. Anderson. Supporting nested locking in multiprocessor real-time systems. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, pages 223–232, July 2012.

[22] B. Ward and J. Anderson. Fine-grained multiprocessor real-time locking with improved blocking. In *Proceedings of the 21st International Conference on Real-Time Networks and Systems*, pages 67–76, October 2013.

A Platform for LEGO Mindstorms EV3 Based on an RTOS with MMU Support

Yixiao Li, Takuya Ishikawa, Yutaka Matsubara, Hiroaki Takada
Graduate School of Information Science, Nagoya University, Japan
{liyixiao,t_ishikawa,yutaka,hiro}@ertl.jp

***Abstract*—LEGO Mindstorms EV3 is the third generation of LEGO's popular programmable robotics kits. However, its standard development environment is not suitable for developing complex programs with real-time requirements. In this paper, a platform for developing real-time applications for EV3 is presented. The platform is based on TOPPERS/HRP2, a static RTOS kernel with MMU support. An algorithm for generating MMU translation tables statically and a method for optimizing TLB flushing are described. The architecture of the platform and some details such as APIs provided and how to reuse Linux device drivers on HRP2 are also introduced. A sample program for controlling a self-balancing robot is developed for comparison with the standard development environment. Finally, the performance of the platform is evaluated.**

I. Introduction

Mindstorms [1] is a popular series of programmable robotics kits released by LEGO Inc. since 1998. Mindstorms robots have been used as important tools in graduate level researches [5] and college education such as computer science [2], [3], [4] for many years. They include a programmable brick computer called Intelligent Brick that controls the whole system, and a set of modular sensors, motors and LEGO blocks to allow users to build robots flexibly.

LEGO Mindstorms EV3 [1] released in 2013 is the third generation of Mindstorms robots. It is equipped with a 300MHz ARM9 CPU, 64MB RAM and supports wireless technologies such as Bluetooth. The standard development environment of EV3 consists of an integrated development environment (IDE) and a Linux-based firmware. The IDE uses a graphical data flow programming language whose developing is done by dragging and dropping icons into a line in order to form commands. See Appendix A for a screenshot of the IDE. All programs developed with the IDE are executed on a virtual machine. The virtual machine is included in the Linux-based firmware running on EV3. The standard development environment is very friendly to beginners who are not familiar with computer programming. However, there are some disadvantages for the developers who are already familiar with common programming languages such as C or C++ as described below:

1) Hard to write complex programs with the graphical programming language
2) Lack of real-time multi-tasking support
3) Virtual machine causing poor real-time performance
4) Proprietary software which is hard to extend
5) Linux-based firmware taking too long to boot up

In this paper, we present **EV3PF** [1], a development platform for those developers who are suffering from the disadvantages mentioned above. The main features of our platform are listed as follows:

1) Booting up very quickly
2) An RTOS kernel supporting protection functionality
3) Programming with common languages such as C/C++
4) Easy-to-use APIs with high real-time performance
5) Open source and an open architecture that third party devices can be easily supported
6) Features to assist the development such as viewing task logs wirelessly via Bluetooth

The rest of the paper is organized as follows. An overview of the TOPPERS/HRP2 kernel which our platform is based on, and details of how to support an ARMv5 MMU such as an algorithm for generating translation tables and a method for optimizing the TLB flushing are described in Section II. In Section III, we present our platform architecture and explain some details of device drivers and APIs. Section IV uses a sample program to show how to develop with our platform and compares it with the standard development environment. We evaluate the performance of our platform in Section V. Finally, the paper is concluded in Section VI.

II. The TOPPERS/HRP2 Kernel

Our platform is based on an RTOS kernel called TOPPERS/HRP2 kernel (High Reliable system Profile version 2) [6]. HRP2 is a static RTOS kernel with memory protection support, which can satisfy high reliability and safety requirements of large-scale embedded systems like EV3. Although there are other RTOS kernels follow a static tailoring approach, such as those based on the OSEK/AUTOSAR automotive standards [16], [17], only few of them do actually support memory protection with static configuration. SAFER SLOTH is an example of those RTOS kernels that support memory protection with a Memory Protection Unit (MPU) [18]. HRP2 supports both MPU and full-blown Memory Management Unit (MMU) which is used by EV3. We ported HRP2 to AM1808, the processor of EV3, since it had not been supported. Most of the porting work is trivial, except supporting a new MMU for HRP2. In this section, we discuss the protection functionality of HRP2 and how to implement it. We begin with an overview of HRP2 kernel in Section II-A. In Section II-B, we show how to support the MMU of AM1808. Finally, we propose a way to optimize TLB flushing in Section II-C.

[1]The source code is available at: http://www.ertl.jp/~liyixiao/ev3pf.tar.xz

A. An Overview of HRP2 Kernel

HRP2 is a static RTOS kernel whose objects (or resources) are defined in and generated from configuration files statically. It supports the protection functionality such as memory protection, object access protection and extended service call functionality. Some key concepts of HRP2 will now be explained.

Kernel Object: A *kernel object* is a system resource managed by the RTOS kernel. Tasks, semaphores and memory areas are typical examples of *kernel objects*. Each type of *kernel object* has four kinds of operations, *type 1 operations*, *type 2 operations*, *management operations* and *reference operations*. In the case of semaphores, *type 1 operations* are for signaling, *type 2 operations* are for waiting, *management operations* are for configuring access rights, and *reference operations* are for acquiring status. The access rights of each kind of operations for a *kernel object* can be configured individually.

Protection Domain: *Protection domains* are disjoint sets of *kernel objects*. There are two types of *protection domains*, the *kernel domain* and *user domains*. Only one *kernel domain* exists and there can be multiple *user domains*. Tasks belonging to the *kernel domain*, called system tasks, are executed in privileged mode and have full access rights to all *kernel objects*. Tasks belonging to a *user domain*, called user tasks, are executed in non-privileged mode with limited access rights. A *user domain* is the finest granularity to grant access rights of a *kernel object*, which means that user tasks in the same *user domain* have the same access rights. Although a task must belong to some *protection domain*, some kinds of *kernel objects* like semaphores can be shared by all *protection domains* without belonging to one of them.

Service Call: The term *service call* in HRP2 has the same meaning as *system call* in Linux. *Service calls* act as an interface between *user domains* and the kernel. This interface is essential to provide system services for a user task which is usually prevented from directly manipulating the kernel's memory. When a *service call* is called, it is aware of the caller's *protection domain*, so illegal operations can be blocked by checking the access rights. Furthermore, new *service calls*, called *extended service calls*, can be defined. A unique number, called *function code*, must be specified by developers for each *extended service call*. The *function code* is used to call an *extended service call* by the `cal_svc()` API.

Memory Object: A *memory object* represents an area of memory controlled by HRP2. Lots of information is associated with a *memory object*, such as base address, size and *attributes* of that area, and access rights granted to *protection domains*. An *attribute* is a property that always holds regardless of which *protection domain* it is accessed from. For example, if a *memory object* has the *attribute* `TA_NOWRITE`, its memory area cannot be written even by a system task. There are also *attributes* like `TA_UNCACHE` to control cache behavior. *Memory objects* can be defined by developers explicitly with the `ATT_MEM` API, or generated from object files registered by the `ATT_MOD` API. *Memory objects* should never overlap, or an error message will show up.

Configurator: In HRP2, developers statically configure the kernel by writing *configuration files*. See Appendix B for an example of *configuration file*. A *configurator* is a tool to parse these *configuration files*. By using the parsing results as input, the *configurator* interprets *template files* to generate necessary C source files or linker scripts. *Template files* are written in a template language defined by HRP2 for generating files. Most necessary *template files* such as generating source files for *kernel objects* have already been provided by HRP2. However, some target-dependent *template files* like supporting memory protection should be implemented by developers.

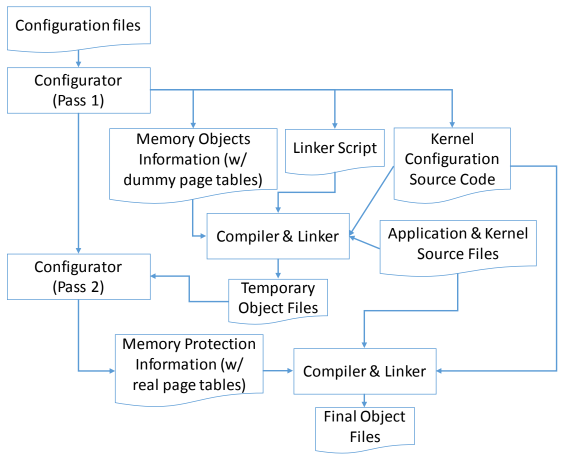

Configuration files
Configurator (Pass 1)
Memory Objects Information (w/ dummy page tables)
Linker Script
Kernel Configuration Source Code
Compiler & Linker
Application & Kernel Source Files
Configurator (Pass 2)
Temporary Object Files
Memory Protection Information (w/ real page tables)
Compiler & Linker
Final Object Files

Fig. 1: Static Configuration Flow

B. Supporting the ARMv5 MMU

Before we explain the supporting of the ARMv5 MMU, we describe the development process with HRP2 kernel. Fig. 1 shows a configuration flow of building the final object files (i.e. kernel image) for HRP2 kernel. HRP2 uses two-pass configuration to generate all source files required. In the first pass, *configurator* generates information of memory objects, a linker script and source code for kernel configuration. Temporary object files will be created for all source files by compiler and linker. The linker script sorts sections of object files by their access rights. All the information of memory objects, such as their base address, size, attributes and access rights, must be determined in this pass. Dummy page tables with the same size of the real ones are generated in this pass to ensure that those information will not change in the second pass. Source code for kernel configuration describes the information to manage RTOS resources, such as initial state and control blocks of tasks. After the first pass is finished, the information of memory protection required by supporting an MMU becomes available. In the second pass, the real page tables will be generated. Finally, the final object files can be created by compiling and linking with the correct memory protection information.

The ARMv5 MMU[8], which is used by the processor of EV3, has not been supported by HRP2. Since HRP2 is a static and single address space operating system, the most difficult and important part of supporting an MMU is to implement the template file to generate translation tables statically. There are two levels of translation tables in ARMv5, to support both sections and pages. The first level translation divides the entire address space into 1MB sized blocks, called sections. Access permissions and cache behavior of a section can be controlled as a whole. A section can also be divided into 4KB sized blocks called pages for fine-grained control. In this case, a section must be associated with a page table for the second level translation.

Each user domain has one section table. The kernel is responsible for configuring MMU to use corresponding section table after switching context. A section table may be associated with multiple page tables. A page table can be shared by multiple section tables when possible for the sake of saving memory space and reducing code size.

We designed an algorithm to generate these section tables and page tables. The input of our algorithm is an array named `MP` generated by the configurator, which contains the information for memory protection. Some important properties of `MP` are listed as follows:

Information of entire address space is included. Each $mp \in$ `MP` represents an area of memory. `MP` is sorted in ascending order of their base addresses for the convenience. mp never overlaps and there is no space between them. That means, the information of every byte in the address space must be contained in some mp, even if it is not used by the kernel.

Base address and size are page-size aligned. The virtual base address, physical base address and size of every $mp \in$ `MP` must be page-size aligned since it is the finest granularity to control the memory by an MMU.

Attributes and permissions are available. Both information of attributes and permissions are required to control cache behavior and set access permissions for a user domain. Let us assume that the attributes can be acquired by the function $\texttt{matr}(mp)$ and the permissions for a user domain can be acquired by the function $\texttt{mper}(mp, dom)$. An mp is $global$ if it has the same access permissions for all user domains. It is $private$ if there is one and only one special user domain which has the different access permissions with others. The function $\texttt{prdom}(mp)$ can be used to get that special domain for a $private$ mp.

Our algorithm consists of four steps, *preprocess*, *generate section tables*, *generate page tables* and *output*, as follows:

Step 1:**Preprocess**. We use an array named `SECTIONS` to store information for all sections. Each $section \in$ `SECTIONS` represents 1MB of memory. For example, `SECTIONS[0]` is the section of memory space from `0x000000` to `0x100000`. $section.val[dom]$ is the value of $section$ in the user domain dom's section table. In this step, we process all sections to generate enough information for next step as follows:

Algorithm 1 Preprocess

```
 1: for all $sec \in$ SECTIONS do
 2:     $ovpset \leftarrow \{mp : mp$ overlaps with $sec\}$
 3:     $glbset \leftarrow \{mp : mp$ is $global\} \cap ovpset$
 4:     $prvset \leftarrow \{mp : mp$ is $private\} \cap ovpset$
 5:     if $|\texttt{ovpset}| = 1$ then        ▷ Suppose $ovpset$ is $\{mp\}$
 6:         for all $dom \in$ USERDOMAINS do
 7:             Generate $sec.val[dom]$ from $mp$
 8:         end for
 9:     else                    ▷ $sec$ needs at least one page table
10:         if glbset=ovpset then
11:             Mark $sec$ as $global$
12:         else if $|\{\texttt{prdom}(mp) : mp \in prvset\}| = 1$ then
13:             if $prvset = ovpset \setminus glbset$ then
14:                 Mark $sec$ as $private$
15:                 $sec.prvdom \leftarrow \texttt{prdom}($any $mp \in prvset)$
16:             end if
17:         end if
18:     end if
19: end for
```

Step 2:**Generate section tables**. Values of all sections that are not associated with any page table have already been generated in last step. If $section$ is $global$, only one page table should be allocated and can be shared by all user domains. If $section$ is $private$, two page tables should be allocated, one for $section.prvdom$ and the other for other domains. Otherwise, every user domain needs a page table for this $section$. We define $\texttt{newpt}(section)$ as a function to allocate a page table associated with $section$. A page table has its $domain$ for generating corresponding access permissions. We define $\texttt{secptval}(pagetable)$ as a function to generate the value representing that a section is associated with $pagetable$. In this step, all values of sections will be generated as follows:

Algorithm 2 Generate section tables

```
 1: for all $sec \in$ SECTIONS do
 2:     if $sec$ is $global \vee sec$ is $private$ then
 3:         $ptshared \leftarrow \texttt{newpt}(sec)$
 4:         $ptshared.domain \leftarrow$ any $dom \neq sec.prvdom$
 5:     end if
 6:     for all $dom \in$ USERDOMAINS do
 7:         if $sec$ is $global \vee (sec$ is $private \wedge dom \neq sec.prvdom)$ then
 8:             $sec.val[dom] \leftarrow \texttt{secptval}(ptshared)$
 9:         else
10:             $pt \leftarrow \texttt{newpt}(sec)$ ▷ Allocate a new page table
11:             $pt.domain \leftarrow dom$
12:             $sec.val[dom] \leftarrow \texttt{secptval}(pt)$
13:         end if
14:     end for
15: end for
```

Step 3:**Generate page tables**. In last step, we have allocated page tables and assigned $section$ and $domain$ for them. According to the first two properties of `MP`, for each entry in a page table, there is one and only one mp that overlaps with it. And by the third property of `MP`, attributes and permissions for that entry can be acquired by $\texttt{matr}(mp)$

and $\texttt{mper}(mp, domain)$. So in this step, the values of page tables can be generated in a trivial way.

Step 4:**Output**. As all values of section tables and page tables have been generated by above steps, the C source code of them can be outputted easily in this step.

We have implemented this algorithm as a template file to generate translation tables. However, that is not enough for supporting the ARMv5 MMU. There is a cache called translation lookaside buffer (TLB) used by the ARMv5 MMU to improve translation speed. On a context switch between user domains, some TLB entries can become invalid, since the access permissions are different. Not like the ARMv6 TLB[8] that supports a feature called application space identifier (ASID), the entries in the ARMv5 TLB are not associated with any address space. Hence, in that case, TLB has to be flushed. The simplest strategy to deal with this is to completely flush the TLB on each context switch.

C. TLB Flushing Optimization

Flushing the entire TLB on each context switch is very expensive and may have a negative effect for real-time applications. For the software-managed TLBs, selective flushing of the TLB can be implemented for better performance[19]. Although ARMv5 TLB is a hardware managed TLB, it does support an instruction to invalidate a single TLB entry by address. With this instruction, we can optimize TLB flush by only invalidating non-global entries on context switch. Non-global entries are entries in translation tables which may change on context switch, which can be determined statically during generating translation tables.

At first, we stored addresses of non-global entries in an array. On context switch, a loop is used to traverse the array and invalidate every entry in it. This approach can be faster than flushing the entire TLB with a small number of non-global entries. However, it introduces an overhead for loop which makes the time of this method growing fast and exceeding the time of flushing the entire TLB very easily.

To optimize it further, we decide to generate the assembly code for invalidating all non-global entries statically. Since traversing an array is no longer necessary, the overhead of loop can be eliminated. In addition, the instruction pipeline should be more efficient without branches. See Section V for a performance comparison of these TLB flushing methods.

III. Platform Architecture

In this section, we first describe the architecture of our platform in Section III-A. Thereafter, we briefly explain how to reuse Linux device drivers on HRP2 and port an open source Bluetooth protocol stack for our platform in Section III-B. Finally, APIs supported by our platform are shown in Section III-C.

A. An Overview of Platform Architecture

Fig. 2 shows an overview of the platform architecture. Our platform is divided into three layers as follows:

Core Services Layer: This layer is to provide services needed by applications and monitor the running application. It mainly consists of the TOPPERS/HRP2 kernel and device drivers. HRP2 kernel provides RTOS functions and protects our platform from potential defects in applications with its protection functionality. Device drivers are implemented to support various features of EV3, such as pulse-width modulation (PWM) for controlling motors and the speaker. Program code in this layer usually runs in privileged mode since it may wish to manipulate hardware devices directly. All services are provided as service calls which can be called from non-privileged mode. A monitor task is used to control the application to be run. For now, the monitor just starts the application after initializing all services properly. The feature of loading applications dynamically and the operations to stop or restart the application will be added in the future.

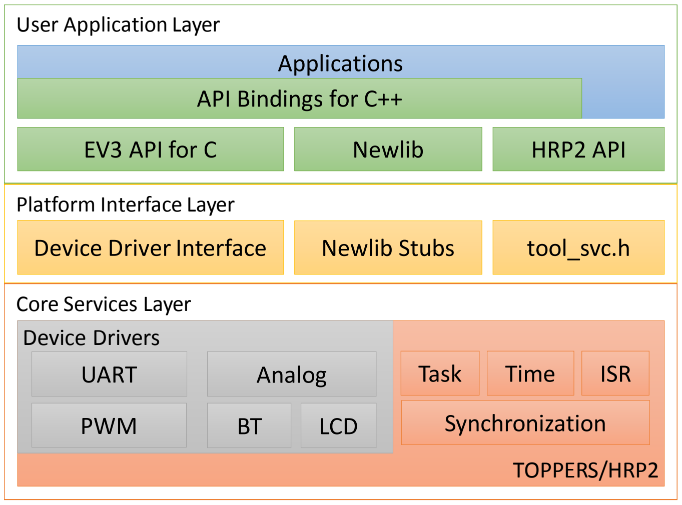


Fig. 2: Architecture of our platform

Platform Interface Layer: This layer acts as an interface between core services layer (CSL) and user application layer (UAL). It specifies all service calls that must be implemented in CSL with their function prototypes. Data structures and macro definitions shared between CSL and UAL are also described. Even if CSL changed a lot, modifications to source code of UAL is unnecessary as long as this layer does not change. Since applications are only dependent on this layer, they can be compiled and linked without any details of CSL. Therefore, binaries built with the same version of platform interface layer are portable, which is very helpful for implementing dynamic loading.

User Application Layer: This layer provides APIs for developing user applications. APIs for C language are provided by default, and it is easy to support other programming languages by implementing API bindings. We have implemented experimental API bindings for C++ to support object-oriented programming with it. Only one application can be run at the same time. All code in this layer runs in non-privileged mode under a determined user domain called `TDOM_APP`.

B. Device Drivers

The biggest challenge we face in implementing device drivers is that the scale of device drivers is too large. The source code of the stock Linux-based firmware has been released by LEGO [13]. We analyzed it with CLOC (Count Lines of Code), a statistics utility to count lines of code. The results shows that there are approximately 60,000 lines of code for device drivers exclusive for EV3 such as UART sensor driver, and approximately half a million lines of code for generic device drivers like Bluetooth protocol stack. Implementing these device drivers from scratch is extremely difficult and will take too much time, if not impossible. To reduce the workload of developing, we must reuse the existing source code such as the Linux-based firmware or other open source projects that can be applied to our platform as possible. We propose an approach to reuse the kernel-space Linux device drivers on HRP2 kernel and evaluate its effectiveness. We also port an open source Bluetooth protocol stack called BTstack [11] to our platform for supporting wireless communication.

1) Reusing Linux device drivers: Many OS research projects have reused code from existing systems to reduced the startup cost. For example, the OSKit[20] presents a framework to reuse C-based components from existing environments. The OSKit defines some libraries such as kernel support library, which should be implemented by developers. With those libraries implemented, a device driver from Linux or FreeBSD can be reused easily by writing some glue code. Although we think the OSKit is too complex for our RTOS, we does adopt a similar and simpler approach to reuse Linux device drivers.

Linux device drivers can be divided into two types, the user-space device drivers and the kernel-space device drivers. The user-space drivers are very difficult to reuse on HRP2 since the APIs[10] they can use are too complex for an RTOS kernel to support. On the other hand, the kernel-space device drivers are developed with the Linux kernel API [9] which mainly provides basic management functions for the kernel. The similarities between the Linux kernel API and the TOPPERS/HRP2 API give us a chance to reuse the kernel-space device drivers. We implemented some functions of Linux kernel API which enable the core parts of drivers to be reused to work. See Appendix D for implementation details.

We now describe the methodology to reuse a Linux driver with an example shown below. At first, the header file `driver_common.h` is included, which contains the Linux kernel API we implemented. Then, some hacks are used to make the driver compiled and working properly. The original source file of the driver is included after the hacks. Unneeded code in that file is commented out. At last, the interfaces of this driver are implemented. In Linux, drivers use the standard file operation system calls as interfaces to communicate with applications. We must adapt implementation of these system calls into corresponding service calls defined in the platform interface layer of our architecture. This work can be done easily by wrapping the file operation functions as extended service calls.

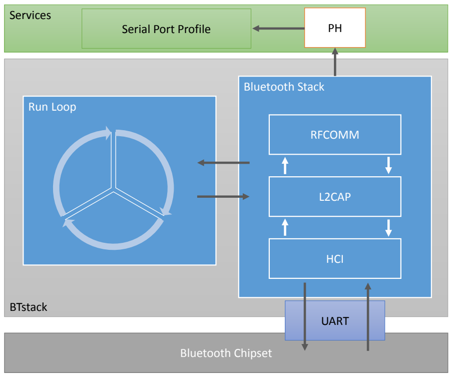


Fig. 3: The architecture of BTstack

```
// Include the common part for a driver
#include "driver_common.h"

// Hacks for this module
#define InitGpio PWM_InitGpio
static void SetGpioRisingIrq(...) {
     ...
}

// Include the source file to reuse
#include "d_pwm.c"

// Interfaces
void pwm_command(...) {
     ...
     Device1Write(...);
     ...
}
```

See Appendix F for a list of Linux device drivers which are reused successfully. We used CLOC to evaluate the effectiveness of our approach. The results show that we reused 14,254 lines of code by writing only 669 lines of code. This approach has saved more than 90% of the coding work for these device drivers.

2) Porting the BTstack: BTstack is an open source Bluetooth protocol stack for embedded systems[11]. Since it can even work without an OS, we believe it can be ported to our platform easily. The architecture of BTstack is shown in Fig. 3. We implemented the hardware initialization for the Bluetooth chipset by referencing the Linux device driver. The chipset and BTstack are communicating with each other via a UART port. We implemented the method stubs in the UART hardware abstraction layer defined by BTstack. BTstack uses a run loop to handle incoming data and schedule work. By default, BTstack only provides two types of run loops, one for POSIX system and the other one for OS-less system. The run loop for OS-less system runs as a busy loop. We modified it to run as a periodic task in our platform. At last, we implemented the packet handlers (PH) to provide the Bluetooth Serial Port Profile (SPP) which can emulate a serial port wirelessly.

It should be noted that the Bluetooth can work at a high data rate up to 3 Mbps. Handling the data transferring with interrupts may have a negative effect for real-time applications. We decided to handle it with a low priority task working in polling mode.

C. Application Programming Interface

There are three types of APIs which can be used by developers to write programs for our platform.

TOPPERS/HRP2 kernel API: This API allows developers to use services provided by HRP2 kernel, such as task synchronization functions. Details of this API are described in the TOPPERS new generation kernel specication[6].

C standard library: The API of C standard library is available, which makes development for our platform as easy as writing a normal C application. Newlib[14], a C standard library implementation has been ported to our platform to support the API. Furthermore, a special file descriptor `SIO_BT_FILENO` for Bluetooth is defined. Developers can do serial communications via Bluetooth simply by calling functions like `fprintf()` and `fgetc()` with `SIO_BT_FILENO`. The use of Newlib requires the dynamic management of memory to support most of its functions. Since memory areas in our platform are configured statically, we statically allocate a fixed-sized memory pool managed by Newlib's memory allocator.

EV3 API for C language: Features exclusive for EV3 such as ultrasonic sensor, servo motors and LEDs are supported by this API. See Appendix C for a list of functions provided by the EV3 API.

IV. A Sample Program for Self-balancing Robot

Developing an application for our platform involves the following steps:

1) Write code with APIs described in Section III-C
2) Build the binary image `hrp2` by the `make` command
3) Generate the boot image `uImage` from `hrp2` by the `mkimage` command
4) Copy `uImage` into root directory of a microSD card
5) Insert the microSD card into Mindstorms EV3
6) Power on EV3 and the application will get executed

To validate the implementation of our platform, we wrote a sample program for a self-balancing two-wheeled robot, which has real-time requirements. See Appendix E for the construction of our robot. We implemented the self-balancing algorithm by referencing the HTWay robot[12]. There are two tasks in our sample program. One task works in a high priority to perform the self-balancing algorithm. The other task works in a lower priority to communicate with user via Bluetooth. It outputs the value of gyro sensor continuously and a user can control the speed and direction remotely. We have tested it and both the tasks can work flawlessly.

On the other hand, the standard development environment of EV3 also includes a sample program for self-balancing called GyroBoy. By default, GyroBoy can keep the balance correctly. However, the standard development environment does not support priority-based scheduling. If we add a greedy task shown in Fig. 4 to log the value of gyro sensor simultaneously, the robot will fall down very easily. This comparison shows that our platform is more suitable for developing the applications with real-time requirements.

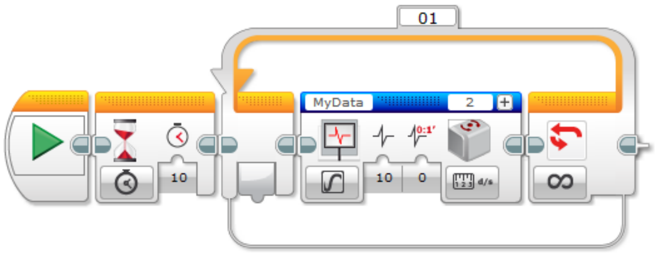


Fig. 4: A task to log the value of gyro sensor in the standard development environment

V. Performance Evaluation

In this section, we describe some brief evaluations of our platform. AM1808, the processor of EV3 is an ARM926EJ-S processor running at 300 MHz[7]. We use performance counters embedded within it to measure the performance of our platform. In addition, we also measure the performance of the Linux-based firmware used by the standard development environment of EV3. Of course, we cannot compare these platforms directly with the measurement results since there are too many differences between them. However, we do believe that the comparison can give us an idea of their relative performance.

As we have mentioned in Section II-B, each protection domain has its own section table. That is, context switching between tasks in the same protection domain will not invalidate any TLB entry, which is similar to thread switching in Linux. This kind of switches can be performed very quickly both in our platform and in Linux. On the other hand, context switching between tasks in different protection domains is just like process switching in Linux, which may cause many TLB entries to become invalid.

We measure the context switching time between tasks in different protection domains to figure out how the method of TLB flushing affects it. The context switching time in our evaluation is from the point that a service call causing the context switch is issued to the point that the first instruction in the new task is executed. The results of three methods which we have described in Section II-C are shown in Fig. 5. A context switch with flushing the entire TLB always costs about 3,500 CPU cycles. If only non-global TLB entries are flushed on context switching, by a loop, the time can be reduced to about 2,530 CPU cycles in the case of three non-global pages. However, if there are more than 28 non-global pages, it becomes slower than the original method. For the optimized method we proposed, which eliminates the overhead of loop, it only takes about 2,200 CPU cycles in the case of three non-global pages. Moreover, it grows much slower and will not cost more than the original method until the number of non-global pages exceeds 52. The results have shown that our optimized method can achieve a very low overhead of TLB flushing when the number of non-global pages is not that large. We have also measured the context switching time of the Linux-based firmware by using LMbench[15]. It shows that the average context switching time is about 310 microseconds (93,000 CPU cycles), which is about 25 times slower than our platform.

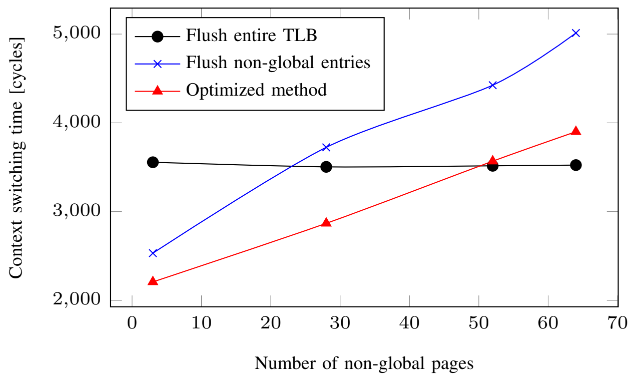


Fig. 5: Context switching time of different TLB flushing methods

TABLE I: Execution time of representative API functions

Function	Average time	Maximum
ev3_uart_sensor_get_raw()	2.96 us	43 us
ev3_motor_set_speed()	4.54 us	40 us
ev3_motor_set_speed()[Linux]	596 us	3424 us

We then measure the performance of some typical APIs provided by our platform. We choose two representative functions `ev3_uart_sensor_get_raw()` and `ev3_motor_set_speed()` from our APIs. For comparison, we implemented `ev3_motor_set_speed()` for the Linux-based firmware of the standard development environment. In our platform, `ev3_motor_set_speed()` can be implemented by simply calling the service call `motor_command()` defined in platform interface layer. In Linux, `ev3_motor_set_speed()` must send a message to the driver by calling the file operation system call `write()`, which can lead to a huge overhead. The measurement results in Table I show that our APIs have much better real-time performance and smaller overhead than the standard development environment.

At last, we measure the memory usage and boot time of our platform. EV3 has 64MB of memory in total. The sample program of our platform described in Section IV takes only 3,276KB in total with 2MB allocated as the memory pool for Newlib. Besides, our platform can boot up in 5 seconds with all devices initialized. On the other hand, the stock Linux-based firmware uses 61,080KB of memory and takes more than 35 seconds to boot up.

VI. Conclusions and Future Work

In this paper, we presented an RTOS-based development platform with protection functionality for LEGO Mindstorms EV3. We ported the TOPPERS/HRP2 kernel to the processor AM1808 used by EV3. An algorithm for generating translation tables statically has been designed to support the ARMv5 MMU on HRP2. We also proposed an optimized method for TLB flushing to reduce the context switching overhead. The optimized method statically generates code for flushing TLB entries which may become invalid on a context switch. The layered architecture, device drivers and APIs of our platform have also been described. To reduce the workload of implementation, we proposed an approach to reuse the Linux kernel-space device drivers on HRP2. We also ported an open source Bluetooth protocol stack called BTstack. A sample program for a self-balancing robot has been developed. By a comparison between this sample program and the one provided by standard development environment, our platform has been proven to be more suitable for real-time applications. At last, we evaluated the performance of our platform. The measurement results have shown that our optimized method of TLB flushing is effective and our platform can boot up much more quickly and provide far better real-time performance than the standard development environment.

As a future work, we plan to support dynamic loading of applications at first. For now, every time the application changes, developers are required to remove the microSD card from EV3, copy the new binary file to it and insert the microSD card again. This duplication of effort can waste so much time and should be eliminated. The architecture of our platform is designed to support this feature. Applications on our platform can already be compiled into individual executable files independent from the core services layer. Future work will focus on designing and implementing a dynamic loader for HRP2, which is a static RTOS kernel.

References

[1] LEGO Group: LEGO.com Mindstorms, http://www.lego.com/en-us/mindstorms/

[2] Fagin, B. S., Merkle, L. D., and Eggers, T. W.: Teaching computer science with robotics using Ada/Mindstorms 2.0, *ACM SIGAda Ada Letters*, Vol. 21, No. 4, pp. 73-78 (2001).

[3] Klassner, F.: A case study of LEGO Mindstorms' suitability for artificial intelligence and robotics courses at the college level, *ACM SIGCSE Bulletin*, Vol. 34, No. 1, pp. 8-12 (2002).

[4] Sell, R., and Seiler, S.: Combined Robotic Platform for Research and Education, *Proceedings of SIMPAR*, pp. 522-531 (2010).

[5] Larsen, K. G., Pettersson, P. and Yi, W.: UPPAAL in a nutshell, *International Journal on Software Tools for Technology Transfer (STTT)*, Vol. 1, No. 1, pp. 134-152 (1997).

[6] TOPPERS Project Inc.: TOPPERS/HRP2 kernel, http://www.toppers.jp/en/hrp2-kernel.html

[7] Texas Instruments Inc.: *AM1808/AM1810 ARM Microprocessor Technical Reference Manual* (2011).

[8] ARM Ltd.: *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition* (2012).

[9] Linux Kernel Organization: Documentation for Linux kernel, https://www.kernel.org/doc/

[10] The Open Group: The Single UNIX Specification, Version 4, http://www.unix.org/version4/

[11] BlueKitchen: BTstack, https://code.google.com/p/btstack/

[12] HiTechnic Products: HTWay - A Segway type robot, http://www.hitechnic.com/blog/gyro-sensor/htway/

[13] LEGO Group: LEGO MINDSTORMS EV3 source code, https://github.com/mindboards/ev3sources/

[14] Newlib, a C standard library for embedded systems, https://sourceware.org/newlib/

[15] MCVOY, Larry W., et al. lmbench: Portable Tools for Performance Analysis, In *USENIX annual technical conference*. 1996. p. 279-294.

[16] OSEK/VDX Group: OSEK/VDX Operating System Specification 2.2.3, http://portal.osek-vdx.org/files/pdf/specs/os223.pdf

[17] AUTOSAR: AUTOSAR Specification of Operating System 3.0.2, http://www.autosar.org/download/AUTOSAR_SWS_OS.pdf

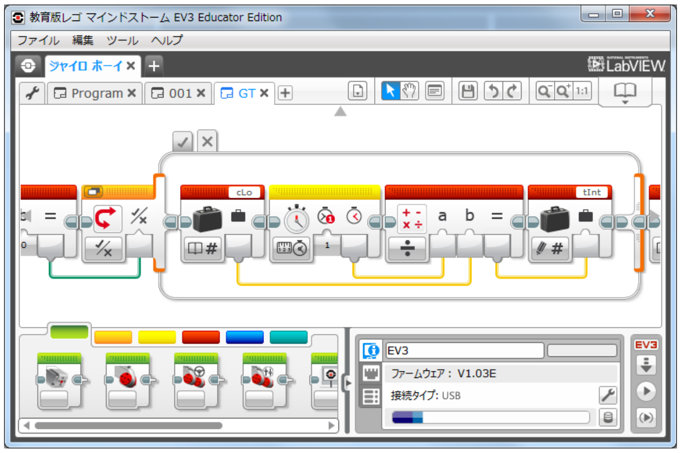
[18] Daniel Danner, Rainer Mller, Wolfgang Schrder-Preikschat, Wanja Hofer, Daniel Lohmann: Safer Sloth: Efficient, Hardware-Tailored Memory Protection, *Proceedings of the 20th Real-Time and Embedded Technology and Applications Symposium*, RTAS 2014.

[19] Nagle, D., Uhlig, R., Stanley, T., Sechrest, S., Mudge, T., Brown, R.: Design tradeoffs for software-managed TLBs, *ACM*, Vol. 21, No. 2, pp. 27-38 (1993).

[20] Ford, B., Back, G., Benson, G., Lepreau, J., Lin, A., Shivers, O.: The Flux OSKit: A substrate for kernel and language research, *ACM*, Vol. 31, No. 5, pp. 38-51 (1997).

Appendix A
The Standard Development Environment for Mindstorms EV3

This is a screenshot of standard development environment and its graphical data flow programming language.



Appendix B
An Example of Configuration File

```
KERNEL_DOMAIN {
  /* create a system task */
  CRE_TSK(MAIN_TASK,{TA_ACT,0,main_task,6,
    1024,NULL});
  /* register a memory object */
  ATT_MOD("sample_kernel.o");
  /* define an extended service call */
  DEF_SVC(SVC1,{TA_NULL,svc1_entry,64});
}
/* a user domain named DOM1 */
DOMAIN(DOM1) {
  /* create a user task */
  CRE_TSK(TASK1,{TA_ACT,0,task1,6,1024,
    NULL});
  /* register a memory object */
  ATT_MOD("sample1.o");
}
/* a user domain named DOM2 */
DOMAIN(DOM2) {
  /* create a user task */
  CRE_TSK(TASK2,{TA_ACT,0,task2,6,1024,
    NULL});
  /* create a semaphore named SEM1 */
  CRE_SEM(SEM1,{TA_NULL,0,1});
  /* configure access rights of SEM1 */
  SAC_SEM(SEM1,{TACP(DOM2),TACP(DOM1)
    |TACP(DOM2),TACP(DOM2),TACP(DOM2)};
}
/* define a shared memory object */
ATT_MEM({TA_NULL, 0x80000000, 4096});
```

Appendix C
EV3 Application Programming Interface

- `ev3_motor_set_speed(motor,speed)`
  Control the speed and direction of a motor.
- `ev3_motor_brake(motor,floating)`
  Brake a motor.
- `ev3_motor_sync (motorA,motorB,speed,turn_ratio)`
  Steer or synchronize with two motors.
- `ev3_motor_get_counts(motor)`
  Get the angular position of a motor (a.k.a. rotary encoder).
- `ev3_gyro_sensor_get_angle(sensor)`
  Detect the rotation of a robot with a gyro sensor.
- `ev3_gyro_sensor_get_rate(sensor)`
  Measure the angular velocity of a robot with a gyro sensor.
- `ev3_ultrasonic_sensor_get_distance (sensor)`
  Measure the distance to an object with an ultrasonic sensor.
- `ev3_touch_sensor_is_pressed(sensor)`
  Detect whether the button of a touch sensor is pressed.
- `ev3_color_sensor_get_color(sensor)`
  Distinguish between 8 different colors with a color sensor.
- `ev3_uart_sensor_get_raw(sensor)`
  Read the raw value of a UART sensor.
- `ev3_led_set_color(color)`
  Set the color of LED on the body of Mindstorms EV3.
- `ev3_button_set_on_clicked (button,handler,exinf)`
  Register a handler for the button click event.

Appendix D
Linux Kernel API Implemented with TOPPERS/HRP2 API

The functions of Linux kernel API implemented in our platform are listed as follows:

1) Kernel-space memory management functions
2) Semaphore API
3) Spinlock API
4) High-resolution timer API

The methodology to implement these functions with TOPPERS/HRP2 API are explained as follows:

Kernel-space memory management: Functions such as `kmalloc()` and `kfree()` are implemented. Linux uses multiple virtual address spaces while the HRP2 kernel uses a single address space model. Therefore it is unnecessary to perform the address translation between kernel space address and user space address. In the case of `kmalloc()`, it can be implemented easily by wrapping the `malloc()` function.

Interface for interrupt handling: Functions such as `request_irq()` which is used to register an interrupt handler for an interrupt number, `request_gpio_irq()` which is used to register an interrupt handler for a GPIO pin, and `free_irq()` which is used to remove an registered interrupt handler are implemented. It should be noted that the dynamic creation of an interrupt handler is not supported by the TOPPERS/HRP2 kernel yet. We chose to define the interrupt handlers statically and let the functions like `request_irq()` or `free_irq()` simply perform enabling or disabling of the corresponding interrupt. The interrupts of GPIO pins are grouped into banks. To implement the `request_gpio_irq()` function, we implemented a GPIO interrupt dispatcher which dispatches interrupt to the corresponding handler.

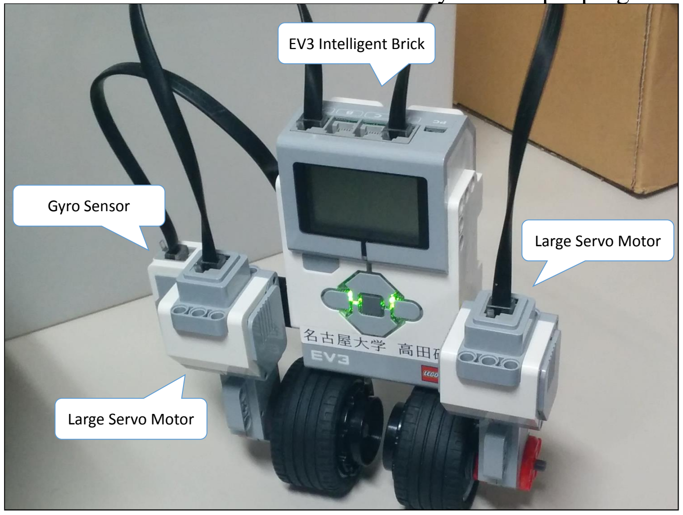
Semaphore API: Functions such as `down_trylock()` and `up()` are implemented. These functions can't be implemented by wrapping the semaphore management functions in the TOPPERS API directly because these operations are not permitted when CPU is locked (i.e. all the interrupts are masked) according to the TOPPERS specification. We implemented these functions natively by referencing the implementation of corresponding functions in the TOPPERS API.

Spinlock API: HRP2 provides the `SIL_LOC_INT()` and `SIL_UNL_INT()` macros to control whether all the interrupts are masked. Functions such as `spin_lock_irqsave()` and `spin_unlock_irqrestore()` are implemented by simply wrapping these macros.

High-resolution timer API: The high-resolution timers have become the standard time framework in Linux since version 2.6.16. The Linux device drivers for Mindstorms EV3 use the high-resolution timer API for handling events periodically. Unfortunately, the HRP2 kernel has not supported the high-resolution timer feature yet. The HRP2 kernel handles periodic events by kernel objects called cyclic handlers. The period of cyclic handlers can only be set in milliseconds. However, the HRP2 kernel does allow us to provide high-resolution periodic ticks. By defining the `TIC_NUME` and `TIC_DENO` macros, we can set the period of system ticks to `(TIC_NUME/TIC_DENO)` milliseconds. We decided to set this period to 200 microseconds and implemented an interface for high-resolution cyclic handlers by handling system ticks. We then implemented the high-resolution timer API with this interface.

APPENDIX E
THE CONSTRUCTION OF OUR ROBOT

This photo shows the construction of a self-balancing two-wheeled robot which can be controlled by our sample program.



APPENDIX F
REUSED LINUX DEVICE DRIVERS

1) PWM Control Module
This driver is used to control the power and measure the angular position of motors with pulse-width modulation (PWM).
2) Analog I/O Module
This driver manages communication and controlling the A to D conversion on an input port. The sensor data of analog sensors is also fetched by this driver.
3) UART Device Controller
This driver supports UART Device Communication Protocol for UART sensors.
4) Soft UART Ports Driver
There are four UART sensor ports on EV3. Two of them are hardware UART ports provided directly by the SoC. The others are UART ports emulated by software. This driver supports those soft UART ports.

Usable RTOS-APIs?

Tobias Klaus, Florian Franzmann, Tobias Engelhard, Fabian Scheler, Wolfgang Schröder-Preikschat
Chair of Distributed Systems and Operating Systems
{klaus,franzmann,engelhard,scheler,wosch}@cs.fau.de
Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU), Germany

***Abstract*—We believe that the *Application Programming Interfaces* (APIs) is a commonly ignored but very important property of a *real-time operating system* (RTOS). It should not only be complete i. e., offer all mechanisms needed to implement common real-time systems, but also be easy to use in order to prevent programming errors and make real-time systems more reliable. Sadly there exists only little information about how a usable RTOS API should look like. Therefore this paper aims to give assistance in assessing and designing RTOS APIs. First we give an overview of concepts we expect an RTOS API to offer and introduce criteria we think must be fulfilled for an API to be called 'usable'. Then we examine the widely known API specifications POSIX and OSEK OS as well as the APIs of the RTOSes *FreeRTOS* and *Windows Embedded Compact 7* w. r. t. to these criteria. Finally we discuss possible reasons for the outcome of our examination and we deduce some advice on how to design RTOS APIs.**

Keywords*—*Operating Systems, System software, Real-Time Systems, Application programming interfaces, Ergonomics, Human factors

I. Introduction

In our everyday world, real-time systems are pervasive. Often, laymen may not realize that they are surrounded by real-time systems but without these systems the most mundane things of our industrialized world would not be possible. Industry needs real-time systems for production, many modern aircraft cannot fly without real-time-critical control systems, and not even something as mundane as a modern car's engine would function without an underlying real-time system. People may take something like a mobile telephone for granted, however, even this is a real-time system.

This enumeration reveals how large parts of our civilization depend on real-time systems acting at the right time in the right way. However, what does 'right' mean in this context? Though in general this is a very tough question often demanding intricate domain knowledge about the physical processes and machines involved, for the underlying RTOS it is simple: Do what the application programmer told you to do!

Therefore it is crucial for an RTOS to know precisely what the programmer wants it to do. The 'language' the programmer uses for this purpose is the API of the RTOS. Since misunderstandings between RTOS and programmer can lead to serious damages like plane crashes, we believe it is crucial that RTOS APIs are well designed and straight forward to use. This makes programming real-time systems less error-prone and provides more time for actually testing the application instead of wondering how the API might be used correctly. Moreover an understandable and easily usable RTOS API can seduce programmers to make their decisions explicit in the code instead of expressing them in terms of the implicit temporal order of instructions. This makes the job of analysis or transformation tools much easier, helping to increase the reliability of the real-time system even more. Nevertheless there appears to be little guidance as to what a well designed API looks like and what to take care of when implementing an RTOS.

In this paper we will first establish what the term real-time means, and what semantics an OS has to offer in order to be called 'real-time capable'. From this we will present our criteria for assesing the appropriateness of an API for programming real-time systems. After that we will take a look at some OSes and OS standards to find out how these perform w. r. t. our criteria. We will then discuss the appropriateness of the APIs we examined and explain where we think these succeed and fail at being usable RTOS APIs. From this discussion we try to deduce some short remarks on how to design an RTOS API.

II. Requirements for an RTOS API

In order to assess RTOS APIs w. r. t. their semantics, we first want to present an overview of important properties of real-time systems. Throughout this paper we will be focusing on embedded real-time systems running on microcontrollers. We will derive concepts and semantics we believe an RTOS for this environment should provide to the application programmer.

Most real-time systems interact with their environment using one or more sensors and actuators. The real-time system reacts to external stimuli signified by a sensor value. These stimuli are called *physical events*. The reaction to such an event is called a *task* and produces some kind of result like e. g., a setpoint for an actuator.[1] More complex tasks may be triggered by a set of events combined by *AND*- and *OR-relations*. In contrast to non-real-time critical computer systems, a real-time system has to react to an ever-changing external environment in a timely manner. Events and tasks are therefore attributed with timing information. Events may be *periodic* or *non-periodic*. Periodic events are described by a *period* and a *phase* while non-periodic events are described by their *minimum inter-arrival time*.

Additionally, all real-time critical events have a *deadline* which denotes the latest point in time at which the result of the associated task has to be available. Missed deadlines may make the result of a task unusable or even destroy the real-time system, leading to damage to people and loss of life. A real-time system's tasks are often split into multiple *jobs*, which represent the basic unit of work. In addition to physical events, jobs may have to react to a change in state of the real-time

This work was supported by the German Research Foundation (DFG) under grant no. SCHR 603/9-1.

[1]Note that we distinguish between the abstract notion of a task and its concrete implementation in the form of an OS process, thread, coroutine etc.

software. These state changes are triggered by the real-time application's jobs and are called *logical events*.

A. Mapping of Real-Time Concepts to APIs

Now that we know which abstract model an OS has to conform to in order to be suitable to a real-time environment, we will take a look at the particular semantics an RTOS API has to offer and the concepts an RTOSes may use to implement them. Most RTOSes do not use the concepts of jobs and tasks directly. In principle, jobs and tasks are passive entities that have to be executed by the real-time system. Most RTOS APIs, however, provide active entities like *threads* [1], *coroutines* [2] or *continuations* [3] to allow the real-time application programmers to execute the abstract jobs and tasks they envision. For performance reasons, multiple jobs that share temporal parameters may be executed by the same active entity. In the rest of this section we will first show how RTOS APIs can handle events. Next we will introduce options for dealing with data dependencies in real-time applications and finally we will detail how an RTOS API may manage shared resources.

1) Event Handling: There are two fundamental ways a real-time system can model event handlers. The first option is to create the active entity when the event occurs and start it. With respect to memory consumption this may be preferable since an active entity that does not exist will not consume any memory for its stack. The second option is to use an *event flag*. An already existing and running active entity blocks itself on the flag and waits for the flag to change state. When the event occurs, the detecting entity toggles the flag and the formerly blocked active entity resumes execution. As event flags represent the most fundamental synchronization mechanism, more complex concepts are built on top of them. These include counting mechanisms like *semaphores* [4] and *barriers* [5], which can be used to implement AND-semantics, as well as the more complex *condition variables* [6]. To support complex tasks it is desirable that the API supports waiting on combinations of events e. g., by specifying event masks. In a non-real-time application these mechanisms would already be sufficient, however, in a real-time system we also require an option for deferring the execution of an event handler to a later point in time. Moreover there need to be concepts like reoccurring *timers* to also support periodic events.

The aforementioned mechanisms are already sufficient to handle logical events. Physical events, however, are usually signalled by interrupts, which by their very nature occur *asynchronously*. An *interrupt service routine* (ISR) that is activated by the occurrence of an interrupt will run immediately, and therefore it will violate the priorities assigned to other active entities. The RTOS API must thus provide some way of requesting the execution of a *synchronous* active entity, from the interrupt handler, which will then obey its assigned priority. If the real-time application programmer keeps the ISR as short as possible, a priority inversion that would otherwise lead to missed deadlines may be avoided. For an RTOS API to be appropriate for a real-time system with firm deadlines, where a silently missed deadline might lead to injury or loss of life, the API should also provide some way of finding out if a deadline has been missed and of reacting accordingly.

2) Data Flow: According to Wolfe and Blaza [7], approximately one third of all real-time systems execute some kind of control law to prevent a physical system from leaving its operating point. The most natural way to model these control systems is to represent them by a data flow from sensors through filters and controllers to actuators. Therefore it would be convenient to have some mapping of this data flow to the real-time system. Although in principle it is possible to establish producer-consumer patterns using the mechanisms we presented in the previous section, this may be awkward. A more natural way of modelling data flow is through mechanisms that combine synchronization with copying of data, like message queues, mailboxes, blackboards etc. This approach has the additional advantage that it may also be used for communication between remote processing nodes or to conveniently migrate jobs from one processing node to another.

3) Resource Management: A real-time application always needs some way to interact with its environment. As a consequence, it needs access to physical devices, which can usually only be used by one active entity at a time. An RTOS API must therefore have some way of providing mutually exclusive access to *shared resources*. However, exclusive access may incur uncontrolled priority inversion [8], risking deadline violations. Therefore an RTOS API must supply mechanisms like the *Priority Ceiling Protocol* (PCP), *Deadline Inheritance* (DI) or *Non-Preemptive Critical Sections* (NPCSes) to avoid such behaviour. More complex resource access schemes like the reader-writer-lock may be provided by the RTOS API to coordinate complex resources like system memory.

B. Criteria for Assessing Real-Time Operating System APIs

Although we now know which semantics are necessary to represent real-time systems, we still have to determine how to judge the implementations the RTOS API offers. In this paper we will employ two criteria to this end.

1) Completeness: Henning [9] identifies *completeness* as the most important criterion of API design. In this paper, we will consider an RTOS API complete if it provides at least a representation of real-time properties like periods, phases, deadlines etc., control flow and data flow dependencies, with and without specifiable delay, and resource management that avoids uncontrolled priority inversion.

2) Usability: Completeness of an RTOS API may be enough to build a real-time system, however, we think that this is not enough to build a reliable real-time system. Many real-time systems are used in a firm or hard real-time environment where missed deadlines have serious consequences. An RTOS API is a man-machine interface since it is used to express a human's wishes in a form that can be processed by a machine. We therefore propose to also consider the psychological aspects of an RTOS API.

Raskin [10] identifies two properties of a good man-machine interface. *Modelessness*, which means that the user does not have to remember which state the machine is in to discern what effect an action will have; and *monotony*, which means that there should be exactly one way for the user to achieve some effect. Both requirements aim to reduce the cost of training personnel, developing the OS and application, and to reduce the cost of maintenance. The *locus of control* of users that are provided with multiple ways of achieving some goal will be drawn away from what they are trying to achieve. Instead of solving the problem at hand, they will spend time choosing the right mechanism for doing so. Since average application programmers are no experts in interface design, they will not necessarily choose the best mechanism.

Note, however, that modelessness and monotony do not mean that the user interface may not offer composite mechanisms. Such a requirement, however – which might be the guiding principle of a naive approach to user interface design – is counterproductive. It would encourage programmers to implement mechanisms they frequently use themselves, leading to a myriad of implementations of the same concepts.

Calculating feasible schedules is an NP-hard problem, while determining events and event handlers is a precondition for creating any schedule at all. Therefore it is much easier for a human to just find the necessary events and event handlers instead of calculating the required schedule. In this paper we will therefore focus on RTOS APIs following the event-triggered paradigm. We do not deny that hard real-time systems, whose failure would endanger human lives, should be executed in a time-triggered fashion. However, in our opinion, real-time system designers should develop even hard real-time systems in the event-triggered way and then use supporting tools to transform the event-triggered design into a time-triggered system. See [11] for an example of such a tool.

III. Choice of RTOSes and RTOS API Standards

One goal of this paper is to give an overview of the state of the art in RTOS APIs. We will therefore present two OS API standards and two OSes, all of which are in widespread use in academia and industry [7]. One fundamental decision an OS designer has to make is whether the system should be configured statically or dynamically w. r. t. active entities and resources. The static approach has numerous advantages: A statically configured RTOS will usually require less run-time resources like RAM and processing power, which – even in our times, where cheap 32 bit microcontrollers are on the way of becoming the norm rather than the exception – is an important issue in the design of real-time systems. Also, it is much easier to analyze the real-time properties of a static system, which allows the designer to guarantee firm and hard deadlines. A dynamically configured OS, on the other hand, may be much more flexible at run-time when reacting to seldom circumstances. The rest of this section will give a short overview of each RTOS API and introduce the job execution abstraction of each approach.

POSIX: From a historical perspective, the *Portable Operating System Interface* (POSIX) standard is the most influential standard in the world of dynamically configured OSes. First adopted by the IEEE in 1988 [12], all Unix-like OSes implement at least part of it. Even the Windows NT series of OSes has had POSIX support in the past, and many embedded non-Unix OSes provide a POSIX compatibility layer. The standard has since grown considerably and by now encompasses a profile that targets real-time systems even for microcontrollers without *memory managment unit* (MMU) and filesystem support [13]. POSIX offers two abstractions for active entities, *processes* and *threads*. These differ insofar as each process has its own address space and therefore cannot access another process's memory, while multiple threads may share the same address space. In this paper we will limit ourselves to the POSIX facilities appropriate for real-time systems. We think that it is appropriate to include POSIX in this paper since many embedded operating systems like eCos or RTEMS implement at least part of the standard. POSIX does not provide direct access to the interrupt handling mechanisms of the hardware. Instead, interrupt handling may be mapped to POSIX's *real-time signals* mechanism. These signals are guaranteed to be delivered in the order they are generated, and if a signal is triggered multiple times, it will be delivered exactly as often as it was triggered.

OSEK OS: Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug (OSEK) OS is an API standard for statically configurable OSes. The standard consists of parts that are mandatory as well as optional parts. In this paper we are going to take a look at the mandatory OS standard [14], which provides an abstraction for active entities, and the optional communication standard [15], which specifies a message passing interface. OSEK OS's active entities are called *basic* and *extended tasks*. Basic tasks have run-to-completion semantics and map directly to our concept of abstract tasks. Extended tasks, on the other hand, may yield the processor voluntarily and therefore are more like other operating system's threads. OSEK OS is in widespread use throughout the automotive industry. Although OSEK has been superseded by the newer *AUTomotive Open System ARchitecture* (AUTOSAR) standard, it is still relevant since it is part of this new standard.

FreeRTOS: The free and open source FreeRTOS[2] is a dynamically configured OS with support for a wide range of different processors. FreeRTOS's API is limited to the facilities required by real-time applications. It provides two implementations of active entities. Similar to the OSEK standard, FreeRTOS names its preemptively scheduled threads *tasks*. Additionally, jobs may also be mapped to *coroutines*, which are scheduled cooperatively.

Windows Embedded Compact 7: This OS is a component-based embedded RTOS that has compile-time configurable support for diverse hardware, a *Graphical User Interface* (GUI), touch screen and playback of digital media. In contrast to all other examined RTOSes it demands an MMU but was included nevertheless, since according to [7] it is used quite frequently in industrial applications. Windows Embedded Compact 7[3] configures resources and active entities dynamically and provides three abstractions for the later ones: *Processes* and *threads* are very similar to their POSIX namesakes, while *fibres* are thread-local coroutines.

Unfortunately none of these RTOSes include an API to detect deadline misses as mentioned in Section II-A1. Therefore in this paper we will not examine this requirement, though we think such an API should at least be offered in firm RTOSes. Moreover we will only take the thread abstraction offered by the four presented approaches into account. The reasons for this decision are twofold: 1. Processes guarantee separate memory address spaces and therefore rely on the presence of an MMU. Since the kind of RTOS we have in mind usually runs on a microcontroller and most microcontrollers do not

[2]http://www.freertos.org/

[3]http://www.microsoft.com/windowsembedded/en-us/windows-embedded-compact-7.aspx

come equipped with an MMU, full-blown processes simply are not an option. 2. Coroutines are non-preemptive and therefore increase the chance of an active entity violating the priority of some other active entity. In principle it is possible to build a real-time system with coroutines, however, we think that this makes the programmer's task much harder and is in conflict with our goal of providing a usable API.

IV. Methodology

The goal of our evaluation is to find out in how far the RTOS APIs we haven chosen succeed w. r. t. the criteria of *completeness* and *usability*. To show if an RTOS's API is complete we first grouped the calls provided by the APIs w. r. t. the criteria we established in Section II. After that we determined how many functions have to be called in order to use these concepts correctly. In this assessment we ignored functions that are only required for setup and configuration.

Next we tried to quantify the monotony of the APIs by exploring all choices an application programmer has to make when implementing one of the basic semantics described in Section II-A. To do so we tried to implement each requirement in all ways permitted by the API and created a test case for each option to verify its semantics. The experience we gathered while implementing these proofs of concepts now allows us to also judge the RTOS APIs w. r. t. the criterion of modelessness.

In Section II-A1 we presented two ways of handling events – either by creating an active entity or by waking a preexisting one up. As part of our evaluation we implemented both methods. First we focused on handling events by directly starting active entities. We implemented the activation of jobs from within other jobs to support logical events and splitting tasks into multiple jobs. To support asynchronous i. e., physical events we also considered all ways of activating entities from within an ISR. Second we used synchronization mechanisms like semaphores to inform waiting active entities of events. For both approaches we also implemented snippets that trigger delayed events. A special case of delayed events may be implemented using the *sleep* mechanism since the event is handled by the same control flow that triggers it. In addition to using the sleep mechanism of most APIs, we also implemented delayed activations by other means.

In order to test the *data flow* mechanisms of the APIs, we implemented simple code snippets using the basic building block of data flow modelling, which is transferring one byte from one job to another.

To execute these test cases conveniently and quickly we did not deploy them on embedded hardware. Instead we used x86-64 based desktop computers. For POSIX snippets this was straight forward since the Linux operating system implements the relevant parts of the standard. As the set of mechanisms we used is basic and timing is not crucial for their semantics, we used Linux-based simulators for the other RTOSes: Trampoline [16] for OSEK OS and the FreeRTOS Linux simulator[4] for FreeRTOS. Unfortunately the FreeRTOS simulator does not support all features of the most recent FreeRTOS release, so we had to forgo execution of some snippets. Microsoft provides a virtual machine for testing Windows Embedded Compact 7 applications. As this virtual machine does not support ISRs we could not run the corresponding test cases. Instead we drew our conclusions from the API documentation. Our code snippets and the complementing test framework are provided for reference[5].

Table I. Number of Functions Associated with each API Mechanism

mechanism	OSEK OS	POSIX	FreeRTOS	Windows EC 7
event handling				
thread administration	✔ (2)	✔ (1)	✔ (8)	✔ (3)
interrupt administration	✔ (6)	✔ (11)	✔ (2)	✔ (5)
synchronization				
event flags	✔ (3)	✔ (7)	✔ (5)	✔ (4)
semaphore	✖	✔ (4)	✔ (6)	✔ (1)
condition variable	✖	✔ (4)	✖	✖
barrier	✖	✔ (1)	✖	✖
timer	✔ (2)	✔ (3)	✔ (4)	✔ (2)
OR-combination	✔ (1)	✔ (5)	✔ (2)	✔ (1)
AND-combination	✖	✔ (1)	✖	✖
data flow				
message queue	✔ (3)	✔ (5)	✔ (12)	✔ (3)
Read-write-lock	✖	✔ (7)	✖	✔ (2)
resource management	✔ (2)	✔ (4)	✔ (2)	✔ (3)
relevant functions	18	40	39	23

V. Results

In this section we will present the results of the experiments we introduced in the previous section.

Table I summarizes the concepts present in each RTOS API. The number of related functions is given in brackets if available. The POSIX standard describes all examined mechanisms and therefor is the most featureful API. Next in line is Windows Embedded Compact 7, followed by FreeRTOS and finally OSEK OS.

The number of options the user has when implementing a required real-time concept serves as an indicator for an APIs's monotony. Table II shows this number for all test cases described in Section IV. For RTOSes supporting ISRs we differentiated between triggering events from within an ISR and from regular active entities. In Table II the first number in each column refers to physical events while the second number represents the number of implementations for the logical event mechanism. Although POSIX signals can be interpreted as an abstraction for hardware interrupts, we did not differentiate them from threads since they are an OS service and not a property of the hardware.

Completeness: As every examined API offers at least one way to implement each requirement, we consider all of them complete.

Monotony: None of the examined RTOS APIs is monotonous, since all of them offer multiple ways of achieving the same semantics. Nevertheless, OSEK OS comes quite close to meeting this criterion: the maximum amount of choices for developers to achieve their aim is four. This indicates a very straightforward API compared to the 45 choices imposed on users by POSIX.

[4]http://www.freertos.org/FreeRTOS-simulator-for-Linux.html

[5]https://www4.cs.fau.de/Research/AORTA/perfectRTOSAPI.tar.gz

Table II. Number of Options for Implementing the Requirements

test case	OSEK OS	POSIX	FreeRTOS	Windows EC 7
event handling				
activation	3 / 4	1	1 / 1	1 / 1
delayed activation	2	1	1	1
periodic activation	3	1	1	1
suspend interrupt	3	1	1	3
synchronization				
control flow	3 / 3	45	19 / 19	18 / 17
delayed control flow	2 / 2	24	19 / 19	19 / 19
sleep	2	32	29	25
data dependency	1	4	3	2
resource management	1	2	2	3

Modelessness: Most APIs do quite well when it comes to modelessness. Even inside of ISRs most RTOSes permit the same API calls as in normal control flows. Nevertheless, dynamically configured systems like POSIX tend to violate this criterion since API mechanism often can be configured extensively. Each of these configurations represents a mode since each configuration introduces a different semantics for some API calls. In FreeRTOS different API calls have to be used for the same purpose, depending on the calling context. This serves to confuse the API user without need. Consequently it includes more relevant functions in Table I than POSIX, though it features less API concepts.

VI. Discussion

Though all of the examined RTOSes can be called complete, none of them fulfil both criteria of *usability*. Nevertheless, some seem to do better than others. In this section we want to discuss particular problems of the APIs and if possible make a guess as to why they are designed the way they are.

The most obvious flaw that comes to mind when comparing Table I and Table II w. r. t. POSIX is that it is overloaded with functionality. During the nearly 30 years of its existence, it has been extended and improved again and again, adding new functionality to the API. The resulting extensive API leads to a huge amount of choices a user has to make in order to achieve some intended semantics. Especially the smorgasbord of synchronization mechanisms should be mentioned here. Although not all mechanisms are marked as 'mandatory' in the standard, and thus are not implemented in most embedded POSIX compatible OSes, this is a severe violation of the principle of monotony. Another problem is that POSIX is very configurable and flexible due to its design goal of being easily adaptable to arbitrary OSes. Normally these words bear a positive connotation but w. r. t. to usability of APIs this does not hold true. Configuring an API mechanism means changing its semantics, which introduces modes. From our point of view both problems can be attributed to the fact that POSIX has always been an integrating standard which has to fit many existing systems and provide even more mechanisms. This approach conflicts with designing a clear and usable API.

The FreeRTOS API requires the user to apply different functions for the same purpose depending on the execution context (thread, coroutine or ISR), which is in conflict with our requirement of modelessness. Whenever FreeRTOS application developers are trying to achieve some goal, they first have to make themselves aware which context they are programming in. Only after that can they decide which function to use. This mental overhead combined with the unnecessary amount of functions available for each task hinders efficient use of this API.

In contrast, Windows Embedded Compact 7 seems to follow a rather good general development pattern. Its API may not be perfect but seems to be quite usable. All results in Table I and Table II show that Windows Embedded Compact 7 does better than the other dynamically configurable RTOS APIs (FreeRTOS and POSIX). Microsoft probably benefits from two factors: 1. The development team seems to have a structured and consistent vision of the API they want to offer. This is probably due to the tighter integration of developers and stricter leadership in the project. Although FreeRTOS is marketed by a commercial vendor too, its API seems to have been designed with much less regard for usability issues. This may be a result of feature growth over time and of designing an API matching the internals of the OS instead of the API user's requirements. 2. Since Microsoft's customers licence specific versions of embedded Windows, Microsoft has complete control over the degree of backward compatibility provided. This is very important since backwards compatibility is one of the major reasons APIs erode over time [9]. Not having to provide backward compatiblity enables the vendor to make drastic cuts where necessary. Nevertheless Windows Embedded Compact 7 suffers from the same issue other dynamically configured OSes have: configurable mechanisms imply different working modes for these mechanisms.

Most of the statically configured OSEK OS's API does not suffer from this flaw. It seems that great care has been taken in OSEK's design to avoid modes, and the standard's designers had the *application programmer* instead of the OS implementer in mind when they composed it. This seems surprising since standardization usually only achieves a perpetuation of the status quo. Another excellent decision in the design of OSEK was to target embedded hardware platforms exclusively. This approach has led to a very lean, straight forward and monotonous interface. A critical look at Table II, however, reveals an apparent exception. There are four different ways of activating threads where all other APIs only provide one function. This may seems excessive but the OSEK OS standard aims for a one-to-one mapping of jobs to active entities. Therefore, in order to support complex control flows, different ways of activating threads are necessary (e. g., one handy API call atomically ends the current thread and activates another one).

Regarding modelessness, OSEK fails only in one aspect, and it does so without a pressing need: An OSEK *event* is coupled with the existence of its associated thread, and therefore, if an event occurs while a thread does not exist, the event will be lost. This design decision is detrimental to the API's usability and OSEK's version of the event mechanism in general appears to be defective. It would have been a much better design decision to specify that an event flag exists even while its associated thread does not. Nevertheless, we think that of all APIs we examined in this paper, OSEK OS provides the best usability experience in an embedded context.

The discussion of the different APIs reveals one commonality: The design process is crucial for the usability of an API.

Since the API should be ergonomic for its end-users and their applications, the designers should have these in mind instead of the underlying implementation. Instead of trying to have one API to cover all applications and purposes w. r. t. to usability it seems to be profitable to aim at one core purpose i. e., real-time in our case. Additionally if planning ahead did not work out and already existing interfaces do not work the way they were planned, one should not shy away from breaking backwards compatibility in favour of better API design.

VII. Related Work

In the field of RTOSes only little scientific work seems to have been done w. r. t. the design of usable APIs. Most papers that have been published either take a look at the overall properties of the RTOS, or, they focus on microbenchmarks of individual properties. Almost no work has been done that assesses the RTOS API itself, and the few examples that do, like Timmerman and Perneel [17], use measures like the 'richness' of the API, which conflicts with its usability. Anh and Tan [18] present a relatively comprehensive collection of microbenchmarks for implementations of RTOS APIs but do not examine the usability of the APIs themselves. The preexisting literature thus does not give a usability assessement of RTOS APIs.

In those cases where new APIs for RTOSes are introduced, these are usually intended for model-driven real-time system development. Since one of the main goals of model-driven development is simulating the resulting real-time system, the invented APIs cater to the needs of simulators instead of those of programmers. In some cases this means that the API does not even offer resource management and therefore is incomplete w. r. t. our criteria. Examples of this approach include Hessel et al. [19], Shaout et al. [20] as well as Maeng et al. [21].

As far as we are aware, no published work targeting the real-time domain explores APIs w. r. t. ergonomic properties like modelessness and monotony. So far, the human factor does not seem to have been at the focus of real-time API design.

VIII. Conclusion

In this paper we first presented a minimal set of semantic concepts that we think are necessary to express the needs of real-time applications. We then showed how these concepts can be mapped to RTOS APIs and introduced the criteria of *completeness*, *modelessness* and *monotony* for assessing the usability of RTOS APIs. After that we presented two RTOS API standards and two RTOSes that we had selected as candidates for a usable RTOS API. We examined these w. r. t. our criteria and are now convinced that the *Offene Systeme und deren Schnittstellen für die Elektronik im Kraftfahrzeug* (OSEK) RTOS API comes quite close to an usable API. Furthermore we came to the conclusion that Windows Embedded Compact 7 is a pleasant API but suffers from the requirements the API of a dynamically configurable OS necessarily faces. We also discussed the shortcomings of FreeRTOS and the *Portable Operating System Interface* (POSIX) standard. The designers of FreeRTOS do not seem to have given much thought to usability while POSIX is trying to be adaptable to any OS regardless of the intended use case. It therefore implements many concepts in more than one way, burdening the application programmer with having to decide which part of the API to use.

In the future we intend to present an RTOS API that surpasses OSEK OS and expresses the abstract real-time concepts we introduced in Section II-B more directly.

References

[1] D. R. Cheriton, "Multi-process structuring and the Thoth operating system," Ph.D. dissertation, University of Waterloo, Ontario, Canada, 1978. [Online]. Available: https://cs.uwaterloo.ca/research/tr/1979/CS-79-19.pdf

[2] M. E. Conway, "Design of a separable transition-diagram compiler," *Commun. ACM*, vol. 6, no. 7, pp. 396–408, Jul. 1963. [Online]. Available: http://doi.acm.org/10.1145/366663.366704

[3] P. J. Landin, "A generalization of jumps and labels," in *Report, UNIVAC Systems Programming Research*, 1965.

[4] E. W. Dijkstra, "Cooperating sequential processes," Technische Universiteit Eindhoven, Eindhoven, The Netherlands, Tech. Rep., 1965, (Reprinted in *Great Papers in Computer Science*, P. Laplante, ed., IEEE Press, New York, NY, 1996). [Online]. Available: http://www.cs.utexas.edu/users/EWD/ewd01xx/EWD123.PDF

[5] P. Tang and P.-C. Yew, "Processor Self-Scheduling for Multiple-Nested Parallel Loops," in *Parallel Processing, 1986. Proceedings. International Conference on*, August 1986, pp. 528–535.

[6] P. B. Hansen, "Concurrent programming concepts," *ACM Computing Surveys (CSUR)*, vol. 5, no. 4, pp. 223–245, 1973.

[7] A. Wolfe and D. Blaza. (2013) *2013 Embedded Market Study*. online. UBM plc. [Online]. Available: http://images.content.ubmtechelectronics.com/Web/UBMTechElectronics/%7Ba7a91f0e-87c0-4a6d-b861-d4147707f831%7D_2013EmbeddedMarketStudyb.pdf

[8] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE TC*, vol. 39, no. 9, pp. 1175–1185, 1990.

[9] M. Henning, "API design matters," *Queue*, vol. 5, no. 4, pp. 24–36, May 2007. [Online]. Available: http://doi.acm.org/10.1145/1255421.1255422

[10] J. Raskin, *The Humane Interface – New Directions for Designing Interactive Systems*. ACM Press/Addison-Wesley Publishing Co., 2000.

[11] F. Scheler and W. Schröder-Preikschat, "The RTSC: Leveraging the migration from event-triggered to time-triggered systems," in *13th IEEE Int. Symp. on OO Real-Time Distributed Computing (ISORC '10)*. Washington, DC, USA: IEEE, May 2010, pp. 34–41.

[12] *Portable Operating System Interface (POSIX®) Base Specifications*, ISO Std. 9945, 1988.

[13] *Standardized Application Environment Profile (AEP) - POSIX Realtime and Embedded Application Support*, IEEE Std. 1003.13-2003, 2004.

[14] *OSEK/VDX Operating System Specification 2.2.3*, OSEK/VDX, February 2005.

[15] *OSEK/VDX Communication Specification 3.0.3*, OSEK/VDX, July 2004.

[16] J.-L. Béchennec, M. Briday, S. Faucou, and Y. Trinquet, "Trampoline – an Open Source Implementation of the OSEK/VDX RTOS Specification," in *11th Int. Conf. on Emerging Technologies and Factory Automation (ETFA'06)*. Prague, Tchèque, République: IEEE, Sep. 2006.

[17] M. Timmerman and L. Perneel, "RTOS State of the Art," Dedicated Systems Experts, Tech. Rep., 2005.

[18] T. N. B. Anh and S.-L. Tan, "Survey and performance evaluation of real-time operating systems for small microcontrollers," *IEEE Micro*, 2009.

[19] F. Hessel, V. da Rosa, I. Reis, R. Planner, C. Marcon, and A. Susin, "Abstract RTOS modeling for embedded systems," in *Rapid System Prototyping, 2004. Proceedings. 15th IEEE International Workshop on*, June 2004, pp. 210–216.

[20] A. Shaout, K. Mattar, and A. Elkateeb, "An ideal API for RTOS modeling at the system abstraction level," in *Mechatronics and Its Applications, 2008. ISMA 2008. 5th International Symposium on*, May 2008, pp. 1–6.

[21] J. C. Maeng, J.-H. Kim, and M. Ryu, "An RTOS API Translator for Model-Driven Embedded Software Development," in *Embedded and Real-Time Computing Systems and Applications, 2006. Proceedings. 12th IEEE International Conference on*, 2006, pp. 363–367.

Notes

OSPERT 2014 Program

	Tuesday, July 8th 2014
8:00– 8:30	Registration
8:30–10:00	Keynote talk: *Open-source and Real-time in Automotive Systems: (not only) Linux, (not only) AUTOSAR* *Paolo Gai*
10:00-10:30	Coffee Break
10:30–12:00	Session 1: RTOS Design and Implementation I ARM-based SoC with Loosely Coupled Type Hardware RTOS for Industrial Network Systems *Naotaka Maruyama, Takuya Ishikawa, Shinya Honda, Hiroaki Takada, and Katsunobu Suzuki* Distributed Real-Time Fault Tolerance on a Virtualized Multi-Core System *Eric Missimer, Richard West, and Ye Li* Fast User Space Priority Switching *Alexander Zuepke, Marc Bommert, and Robert Kaiser*
12:00–13:30	Lunch
13:30–15:00	Session 2: Mixed-Criticality Systems Implications of Multi-Core Processors on Safety-Critical Operating System Architectures *Stefan Burger, Kevin Müller, Oliver Hanka, Michael Paulitsch, Andrea Bastoni, Henrik Theiling, and Matthias Heinisch* Towards Hard Real-Time Control and Infotainment Applications in Automotive Platforms *Mian M. Hamayun, Alexander Spyridakis, and Daniel S. Raho* Mixed-Criticality on Multicore (MC^2): A Status Report *Namhoon Kim, Jeremy P. Erickson, and James H. Anderson*
15:00–15:30	Coffee Break
15:30–16:30	Session 3: RTOS Design and Implementation II A Platform for LEGO Mindstorms EV3 Based on an RTOS with MMU Support *Yixiao Li, Takuya Ishikawa, Yutaka Matsubara, and Hiroaki Takada* Usable RTOS-APIs? *Tobias Klaus, Florian Franzmann, Tobias Engelhard, Fabian Scheler, and Wolfgang Schröder-Preikschat*
16:30–18:00	Discussion and Closing Thoughts

	Wednesday, July 9th – Friday, July 11th 2014
	ECRTS main conference.

Max
Planck
Institute
for
Software Systems