Project Number 288008

D 2.2 Concepts for Interrupt Virtualisation

Version 1.0
4 June 2012
Final

Public Distribution

Eindhoven University of Technology and Technical University of Denmark

Project Partners: **AbsInt Angewandte Informatik, Eindhoven University of Technology, GMVIS Skysoft, Intecs, Technical University of Denmark, The Open Group, University of York, Vienna University of Technology**

Project Partner Contact Information

AbsInt Angewandte Informatik Christian Ferdinand Science Park 1 66123 Saarbrücken, Germany Tel: +49 681 383600 Fax: +49 681 3836020 E-mail: ferdinand@absint.com	**Eindhoven University of Technology** Kees Goossens Potentiaal PT 9.34 Den Dolech 2 5612 AZ Eindhoven, The Netherlands E-mail: k.g.w.goossens@tue.nl
GMVIS Skysoft Tobias Schoofs Av. D. Joao II, Torre Fernao Magalhaes, 7 1998-025 Lisbon, Portugal Tel: +351 21 382 9366 E-mail: tobias.schoofs@gmv.com	**Intecs** Silvia Mazzini Via Forti trav. A5 Ospedaletto 56121 Pisa, Italy Tel: +39 050 965 7513 E-mail: silvia.mazzini@intecs.it
Technical University of Denmark Martin Schoeberl Richard Petersens Plads 2800 Lyngby, Denmark Tel: +45 45 25 37 43 Fax: +45 45 93 00 74 E-mail: masca@imm.dtu.dk	**The Open Group** Scott Hansen Avenue du Parc de Woluwe 56 1160 Brussels, Belgium Tel: +32 2 675 1136 Fax: +32 2 675 7721 E-mail: s.hansen@opengroup.org
University of York Neil Audsley Deramore Lane York YO10 5GH, United Kingdom Tel: +44 1904 325 500 E-mail: Neil.Audsley@cs.york.ac.uk	**Vienna University of Technology** Peter Puschner Treitlstrasse 3 1040 Vienna, Austria Tel: +43 1 58801 18227 Fax: +43 1 58801 918227 E-mail: peter@vmars.tuwien.ac.at

Contents

Document Control

Version	Status	Date
0.1	First draft for review	18 May 2012
0.2	Updated draft for review	29 May 2012

Executive Summary

This document describes the deliverable *D 2.2 Concepts for Interrupt Virtualisation* of work package 2 of the T-CREST project, due 9 months after project start as stated in the Description of Work. This document presents the rationale, requirements, and concepts for interrupt virtualisation.

1 Introduction and Rationale

1.1 Real-Time Systems

Real-time systems contain one or more applications, each consisting of multiple communicating tasks. These tasks are distributed over microprocessors that are software programmable. To facilitate developing and writing applications, often a real-time operating system (RTOS) is used on each of the microprocessors. Examples of RTOS are [14, 7, 5, 19, 16, 17, 8, 6, 3, 1, 2]. A RTOS allows multiple tasks to share a single processor, and allocates space ((virtual) memory addresses) and time (percentage of the processor usage) to each of the tasks. In the remainder, we focus on the time-sharing or scheduling of tasks on a processor.

Scheduling may be static (based on design-time information only) or dynamic (also taking into account run-time information). Orthogonally, it may be non-preemptive (a task executes until it finishes or explicitly yields control of the processor to another task) or preemptive (where a task executes for a certain amount of time, before control of the processor returns to the RTOS, which then schedules a next task). Examples of the scheduling algorithm employed by the RTOS are static order, round robin, time-division multiplexing, static or dynamic priority, and so on. The scheduling algorithm is of great importance since it is one of determinants of the response times of the tasks. A scheduling algorithm is work conserving if the processor is never idle when there is a task ready and waiting for execution. This often improves the average response time of tasks, but at the cost of a higher worst-case response time.

Traditionally real-time systems contained a single dedicated application consisting of multiple tasks. Each task could be analysed for its real-time behaviour (e.g. worst-case execution and response times). After they were combined with a predictable (i.e. analysable real-time) scheduler, the timing behaviour of the system could be determined. Predictable work-conserving (often non-preemptive) schedulers are adequate when running a single real-time application. A predictable system therefore allows each application to be characterised in terms of worst-case timing behaviour, usually computed from the behaviour of all tasks in the system and the scheduler.

1.2 Virtual Platforms

More recently, real-time systems often contain multiple distributed applications, whose tasks are sharing processors. Figure 1 illustrates the case where they are all real-time, and a single predictable scheduler manages the tasks of all applications. However, often some of the applications are not real time. These mixed-criticality systems present new additional problems since non-real-time tasks may not have a (known) worst-case execution time. Their co-existence with real-time tasks mean that the worst-case response times of the latter are affected, and perhaps even become unbounded too. As a result, predictable preemptive schedulers are required when multiple real-time and non-real-time applications are supported in the same system. (A preemptive scheduler ensures that tasks can be started independently of the execution or blocking behaviour of other tasks.) Now only each real-time application has a predictable behaviour in terms of worst-case timing, computed from the timing behaviours of its constituent tasks and the preemptive scheduler. Non-real-time tasks usually have no predictable timing behaviour.

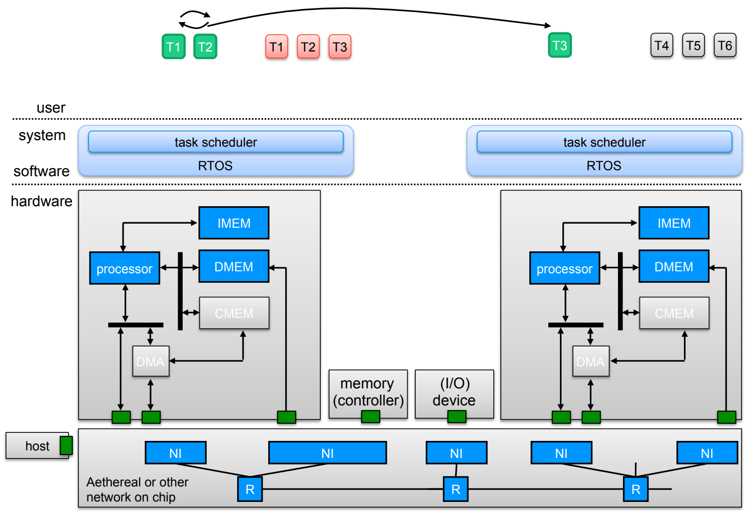

Figure 1: Single-level scheduler RTOS where a single task scheduler manages all tasks, even when of different applications.

Ideally, multiple applications in a single real-time system do not affect each other, in the sense of changing each other's worst-case and actual timing behaviours (execution and response times, jitter). Static non-work-conserving schedulers, such as time-division multiplexing, provide the benefit of timing isolation (a.k.a. partitioning, or composability between tasks), since tasks are scheduled according to a precomputed sequence that does not depend on the run-time behaviour of any application. While the worst-case behaviour of a real-time application is computed as before, from the timing behaviours of its tasks and the scheduler, now also the actual behaviour (as opposed to worst-case behaviour) of the application is independent of other applications. This partitioning method is used in time-triggered architecture such as [17, 15].

Each application can be thought of as executing in its own virtual platform, defined by the scheduler (i.e. the budget for each of its tasks). An application's actual and worst-case timing behaviours are independent from other applications, potentially executing on the same physical platform. As a result, partitioned or composable real-time systems allow applications to be developed and tested independently, and to be integrated iteratively. In other words, circular verification whereby changes (improvements, bug fixes, updates) in one application expose undesired behaviour in another application that then require changes, possibly ad infinitum, is avoided.

Different applications (real-time or not) are developed with different requirements, often by different independent software vendors. As a result, each application may require its own model of computation, e.g. static or dynamic dataflow, Kahn process network, time-triggered, and scheduler, e.g. static order, priority-based, round-robin, time-division multiplexing. This requires a two-level approach to scheduling with *inter-application scheduling* and *intra-application scheduling*, respectively. The inter-application scheduler first decides which application to execute and the intra-application sched-

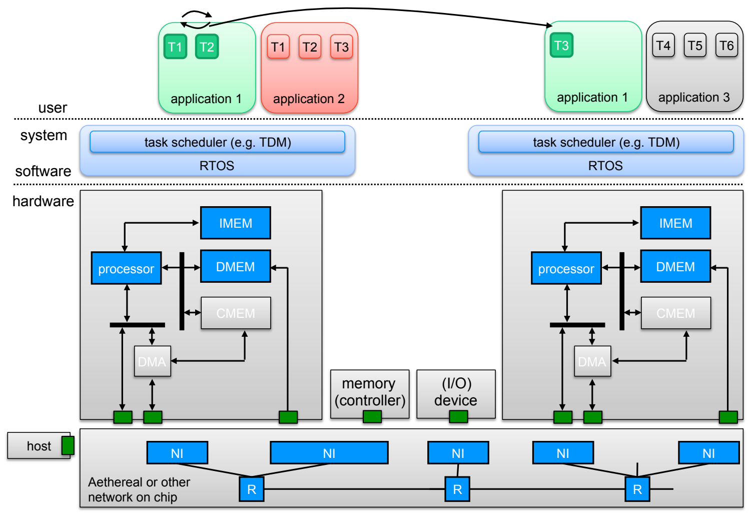

Figure 2: Single-level scheduler RTOS supporting partitioning.

uler then schedules tasks from that application. The purpose of the inter-application level is to partition applications and it must be non-work-conserving and preemptive. The application scheduler is trusted time-critical code, in the sense that any functional or timing error can lead to the entire system, i.e. all applications, malfunctioning.

Second-level task schedulers, on the other hand, are application specific. Since application developers, rather than RTOS or system integrators, specify and write their own task schedulers, they are not trusted time-critical code. For this reason, they execute in application time (and space), and not in system time (and space). A malfunctioning task scheduler only affects the application of which it is a part.

It is possible to remove the second-level intra-application scheduling, and only use the intra-application (TDM) scheduling (see Figure 2), but this is likely to come at the cost of a severe reduction in processor utilisation [20]. Similarly, while some RTOSs implement both partitioning and two-level scheduling (illustrated in more detail below, and [20, 12]), they only allow non-preemptive intra-application scheduling, which is likely to be similarly expensive. Preemptive intra-application scheduling promises to offer better processor utilisation and, more importantly in the T-CREST context, a lower worst-case response time for applications.

Figure 3 illustrates the concepts introduced above. The Compose RTOS [14, 21] and the CompSOC platform are shown, since they implement a number, but not all, of the requirements. Each processor has local instruction and data memories. Additionally each has specialised communication memories with associated DMAs to ensure partitioning [10]. Processor tiles, (I/O) devices, and predictable distributed memories (SRAM and DRAM [9]) are interconnected by a predictable network on chip [13]. The partitioning RTOS essentially consists of a preemptive non-work-conserving

TDM inter-application scheduler. Each application contains its own dedicated intra-application task scheduler. Compose currently only allows non-preemptive task schedulers [20], see Section 3 below.

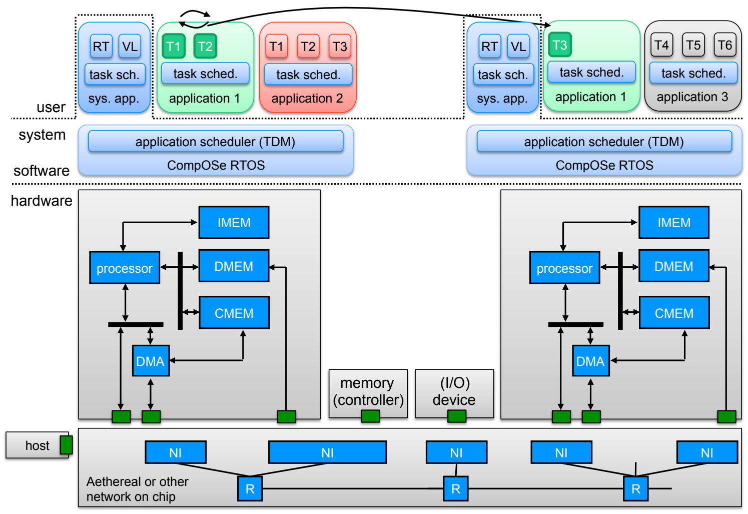

Figure 3: CompSOC and Compose RTOS, illustrating partitioning and two-level scheduling.

1.3 Interrupts

Tasks in a real-time application do not only communicate amongst themselves, but also interact with the physical environment in which the real-time system is embedded. While output can often be scheduled in the future using e.g. timers, asynchronous inputs are either obtained by the processor polling the I/O device or by receiving interrupts from the I/O device. The latter is illustrated in Figure 4, where I/O devices are connected to the processor and the RTOS includes an ISR.

While many operating systems support both options and manage to bound the time to serve the interrupt, the impact on the worst-case execution time of the application running when the interrupt arrives is not trivial to analyse. An interrupt interferes with the execution and timing of an application, since the interrupt service routine (ISR) consumes some of the execution time. This complicates the real-time analysis in all cases, but is even not permitted in the case of partitioned systems, where applications should not interfere with each other at all. Handling an interrupt that is destined for another application has an impact on the current application. Virtualising I/O interrupts such that they are contained in the virtual platform belonging to the application is therefore required.

The rest of this document is structured as follows. Requirements following from our analysis are stated in Section 2. Section 3 reviews the state of the art in RTOSs in terms of characteristics and requirements. Section 4 then introduces the concepts that aim to fullfil the identified requirements. Lastly, conclusions are drawn in Section 5.

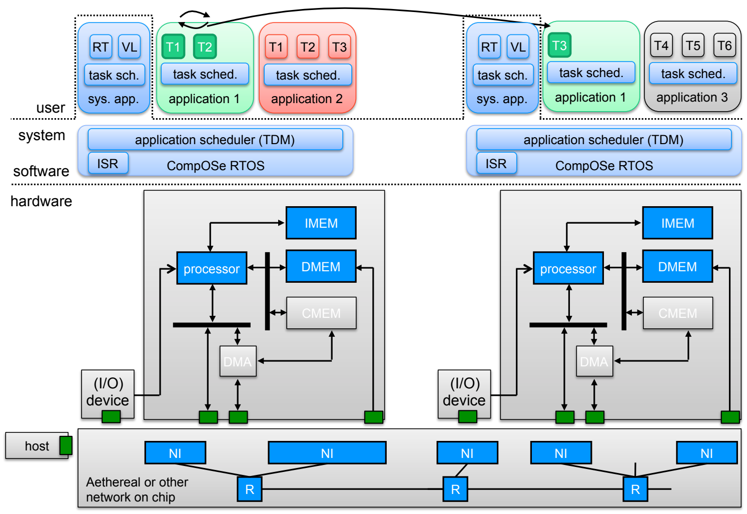

Figure 4: Illustration of interrupt structure with I/O devices connected to processors and an ISR in the RTOS.

2 Requirements

Based on the introduction and rationale, we state the following nine requirements for the virtualised interrupts.

1. The RTOS must support multiple concurrent applications, each consisting of a set of communicating tasks.
2. An application's tasks may be mapped on one or more processors.
3. The RTOS must partition applications, in the sense that the worst-case and actual timing behaviours of any application are not affected by the absence or behaviour of any other application.
4. Each application must specify its own task scheduler, specific to its model of computation.
5. The following models of computation shall be supported: cyclo-static dataflow, Kahn process networks, time-triggered.
6. A task scheduler may be non-preemptive (cooperative) or preemptive.
7. A preemptive task scheduler shall be able to allocate a cycle budget to a task. The task will be preempted in the application when the cycle budget has been depleted.

8. A preemptive task scheduler shall be able to allocate a deadline for a task. The task will be preempted at the latest time before the deadline when the application is active. The deadline shall be given as an absolute time.

9. Each interrupt source, such as a hardware device, shall be assigned to and handled by one application. Interrupt arrival and handling shall not impede partitioning.

3 State of the Art

To ensure that the timing requirements of real-time applications are met, some existing RTOSs do not allow sharing resources at inter- and intra-application levels at all. However, since not sharing resources is expensive or even impossible (think of the single DRAM interface on a chip), much research has been directed this problem. The state of the art in ensuring real-time requirements of applications when they share resources is to isolate applications, and therefore, to prevent interferences that may invalidate the timing requirements. The isolation mechanisms that have been proposed range from high level of isolation that aims to minimise sharing critical resources, to full cycle-level isolation. There are two main categories of approaches to achieve isolation between applications executing on a platform: first, those implementing resource partitioning, and second, those that virtualise shared resources.

The first one is followed by partitioning RTOSs that offer real-time non-work-conserving preemptive inter-partition scheduling. Every application is then assigned to a partition. The level of isolation that a partitioning system provides to the applications may vary from cycle-level to coarser levels. A cycle-level partitioning RTOS is the strictest in the sense that inter-application interference is completely prohibited at cycle-level, and therefore, the application temporal behaviour is fully independent of absence or behaviour of any other application. Thus, mixed time-criticality applications can be easily designed, verified, and integrated on the same platform.

In virtualization approaches, a hypervisor creates number of virtual machines (VMs) to execute multiple commodity OSs on a single platform. The VM-based approaches often partition space (e.g. virtual memory) between applications, but temporal dependencies between the VMs are usually still possible.

The following table compares a number of RTOSs and on their relevant features. In the table, partitioning is defined as real-time non-work-conserving preemptive scheduling. Two-level scheduling is defined as using partitioning between applications, including a separate arbiter per application for task scheduling. Virtual I/O interrupts are defined as interrupts from external sources that only affect the single application to which they are assigned.

Few RTOSs implement partitioning or two-level scheduling, and none except for Compose, PikeOS, INTEGRITY, and LynxOS-178 implement both. The three latter ones are commercial products. PikeOs and INTEGRITY have multi-core versions that are a single copy of the RTOSs for multi-core platforms. PikeOs and LynxOS-178 do not provide any power management mechanisms, while INTEGRITY has a user-mode power management support. All the three RTOSs conform to the ARINC standard [11], which is a specification for time and space partitioning in Safety-critical avionics Real-time operating systems. According to ARINC, processing time is divided into number of time

Table 1: Comparison of RTOS features.

	real time	partitioning	two-level scheduling	preemptive two-level	virtualised I/O interrupts
CompOSe [14, 21]	yes	yes	yes	no	no
TTA [17]	yes	yes	no	no	n/a
μC/OS-II[18]	yes	no	no	no	no
RTEMS [7]	yes	no	no	no	no
PikeOS [5]	yes	yes	yes	yes	no
eCOS [19]	yes	no	no	no	no
OKL4 [16]	yes	no	yes	yes	yes
QNX Neutrino [6]	yes	no	no	no	no
INTEGRITY [3]	yes	yes	yes	yes	no
VxWorks [8]	yes	yes	no	no	no
ERIKA [1]	yes	no	no	no	no
LynxOS-178 [4]	yes	yes	yes	yes	no
FreeRTOS [2]	yes	no	no	no	no
hypervisor	no	no	yes	yes	yes

slices known as the partitions. Every partition is assigned an application. To the best of our knowledge, none of the three provide cycle-level isolation between the isolated partitions, and subsequently, between the applications. None of them, including Compose, implement virtual interrupts with preemptive two-level scheduling.

4 Virtualised Interrupt Concepts

Combining the state of the art and requirements, we propose the following approach and concepts. The Compose RTOS is a good candidate for use in the T-CREST project, since it offers some critical required features, namely predictability (including a design flow for real-time analysis), and cycle-level partitioning (a.k.a. composability). The requirements that it does not satisfy are:

- Requirement 3 The RTOS must partition applications, in the sense that the worst-case and actual timing behaviours of any application are not affected by the absence or behaviour of any other application.
- Requirement 6: A task scheduler may be non-preemptive (cooperative) or preemptive.
- Requirement 7: A preemptive task scheduler shall be able to allocate a cycle budget to a task. The task will be preempted in the application when the cycle budget has been depleted.
- Requirement 8: A preemptive task scheduler shall be able to allocate a deadline for a task. The task will be preempted at the latest time before the deadline when the application is active. The deadline shall be given as an absolute time.
- Requirement 9: Each interrupt source, such as a hardware device, shall be assigned to and handled by one application. Interrupt arrival and handling shall not impede partitioning.

To deal with Requirement 6 it is necessary to distinguish the logical interrupt used by the inter-application scheduler to implement partitioning (non-work-conserving inter-application scheduling) and the logical interrupt used for preemptive intra-application scheduling. It is easy to necessary to multiplex these two distinct logical interrupts on one physical interrupt line and to distinguish them through the use of an interrupt vector. However, a number of open issues remain. First, the architecture to generate the interrupts, in particular the timers (e.g. a single software-multiplexed timer, or a timer per application) requires further investigation. Second, how and where in the architecture are interrupts for applications other than the currently executing interrupt handled? For example, is masking interrupts enough, does each application require its private virtual ISR, or is a single RTOS ISR enough? Finally, ensuring partitioning such that no matter when any (set of) interrupts arrive, it only affects the (timing of the) relevant application. This is captured by Requirement 3. Figure 5 gives a high-level impression of some of these issues.

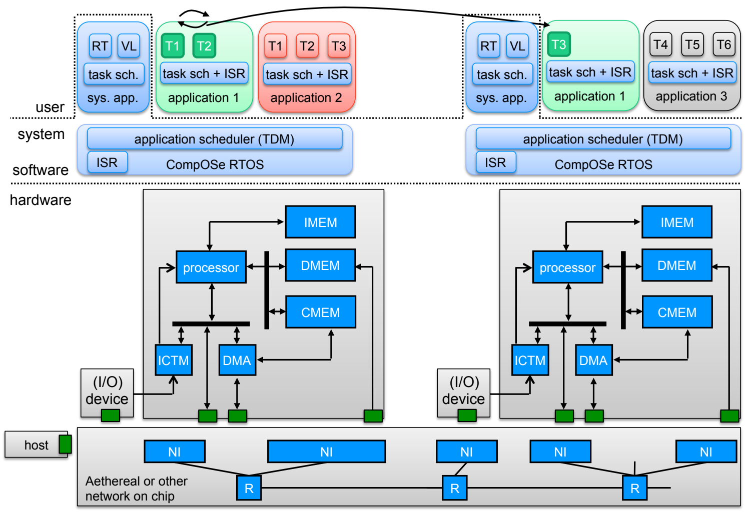

Figure 5: Illustration of virtualized interrupt structure with an ICTM in the processor tile and application-level ISRs.

The implementation of Requirement 9 is likely to be closely related to the infrastructure needed for Requirement 6 since interrupts arriving not from timers under control of the processor, but from asynchronous (I/O) devices must be controlled and released only when the relevant application is active.

Requirements 7 and 8 are related and deal with budgeting (in terms of execution cycles – how much, and in terms of time – when) a virtual interrupt's generation. These will impact the software and possibly hardware architectures, but also require determining how interrupts are budgeted and scheduled in a sparse time base, in the sense that the execution of an application is discontinuous in time.

The concepts for interrupt virtualisation therefore include:

- hardware; an interrupt, clock, and timing manager (ICTM), shown in Figure 5, that accepts, generates, and routes interrupts on the basis of e.g. currently active application, absolute and virtualised time, and frequency and power state of the processor.
- software; in particular, the software stack including system ISR, (privileged?) APIs and drivers to access the virtualised interrupt hardware, and possibly application-level and application-specific ISRs, shown bundled with the task schedulers in Figure 5.
- analysis and tools; the resulting hardware/software architecture must be composable, i.e. applications cannot be affected by each other's behaviours (including interrupt handling), which must be shown through a thorough analysis. Moreover, the architecture must be predictable, i.e. given a configuration of the architecture (i.e. how its components such as arbiters have been programmed), the worst-case execution times of predictable applications must be computed using automated tooling.

5 Conclusions

In this document, we have introduced mixed-criticality real-time systems that interact with the surrounding physical world through interrupts. Nine requirements related to hardware, middleware, software, and mapping, were identified and discussed to enable applications in these systems to be independently designed, verified, and executed. This reduces design & verification phase of system design, shortening the time to market. A survey was then presented of 14 real-time operating systems (RTOS), indicating which of the requirements they can satisfy. We concluded that the CompOSe RTOS, executing on the CompSOC platform, satisfies many of the requirements and is a suitable starting point for this work. This was followed by a discussion on interrupt virtualization that indicates a trajectory towards implementing the missing requirements. This work will be carried out as a part of the T-CREST project.

References

[1] Erika Enterprise. `http://erika.tuxfamily.org/`.

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

[3] INTEGRITY. `http://www.ghs.com/products/rtos/integrity.html`.

[4] LynxOS-178. `http://www.lynuxworks.com/rtos/rtos-178.php`.

[5] PikeOS. `http://www.sysgo.com/products/pikeos-rtos-and-virtualization-concept/`.

[6] QNX-Neutrino. `http://www.qnx.com/products/neutrino-rtos/`.

[7] Real-time executive for multiprocessor systems (RTEMS). `http://www.rtems.com`.

[8] VxWorks. `http://windriver.com/products/vxworks/`.

[9] Benny Akesson and Kees Goossens. Architectures and modeling of predictable memory controllers for improved system integration. In *Proc. Design, Automation and Test in Europe Conference and Exhibition (DATE)*, pages 1–6. IEEE, March 2011.

[10] Jude Ambrose, Anca Molnos, Andrew Nelson, Sorin Cotofana, Kees Goossens, and Ben Juurlink. Composable local memory organisation for streaming applications on embedded MPSoCs. In *Proc. Int'l Conference on Computing Frontiers (CF)*, CF '11, pages 23:1–23:2, New York, NY, USA, May 2011. ACM.

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

[12] Ashkan Beyranvand Nejad, Anca Molnos, and Kees Goossens. A unified execution model for data-driven applications on a composable MPSoC. In *Proc. Euromicro Symposium on Digital System Design (DSD)*, pages 818–822, Washington, DC, USA, August 2011. IEEE Computer Society.

[13] Kees Goossens and Andreas Hansson. The Aethereal network on chip after ten years: Goals, evolution, lessons, and future. In *Proc. Design Automation Conference (DAC)*, pages 306–311, June 2010.

[14] Andreas Hansson, Marcus Ekerhult, Anca Molnos, Aleksandar Milutinovic, Andrew Nelson, Jude Ambrose, and 2010. Kees Goossens, Elsevier. Design and implementation of an operating system for composable processor sharing. *Microprocessors and Microsystems (MICPRO)*, 35(2):246–260, March 2011. Special issue on Network-on-Chip Architectures and Design Methodologies.

[15] Andreas Hansson, Kees Goossens, Marco Bekooij, and Jos Huisken. CoMPSoC: A template for composable and predictable multi-processor system on chips. *ACM Transactions on Design Automation of Electronic Systems*, 14(1):1–24, 2009.

[16] Gernot Heiser and Ben Leslie. The OKL4 Microvisor: Convergence point of microkernels and hypervisors. In *Proceedings of the 1st Asia-Pacific Workshop on Systems*, pages 19–24, New Delhi, India, Aug 2010.

[17] Herman Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer, 2011.

[18] Jean J. Labrosse. *Microc/OS-II*. R & D Books, 2nd edition, 1998.

[19] Anthony Massa. *Embedded Software Development with eCos*. Prentice Hall Professional Technical Reference, 2002.

[20] Anca Molnos, Ashkan Beyranvand Nejad, Ba Thang Nguyen, Sorin Cotofana, and Kees Goossens. Decoupled inter- and intra-application scheduling for composable and robust embedded MPSoC platforms. In *Workshop on Mapping of Applications to MPSoCs (MAP2MPSOC)*, May 2012.

[21] Andrew Nelson, Anca Molnos, and Kees Goossens. Composable power management with energy and power budgets per application. In *Proc. Int'l Conference on Embedded Computer Systems: Architectures, MOdeling and Simulation (SAMOS)*, pages 396–403, July 2011.