Temporal and Spatial Partitioning of a Time-Triggered Operating System based on Real-Time Linux

R. Obermaisser
Vienna University of Technology
romano@vmars.tuwien.ac.at

B. Leiner
TTTech Computertechnik AG
bernhard.leiner@tttech.com

Abstract

Real-time Linux variants are becoming prominent solutions for the development of embedded systems. Compared to traditional real-time operating systems, embedded system engineers can leverage solutions and knowhow from the Linux development community (e.g., development tools, applications, drivers). Due to the availability of implementations of Internet protocols and network drivers, Linux also facilitates the implementation of embedded systems connected to the Internet. The goal of this paper is to evaluate experimentally the capabilities of the Real-time Linux variant RTAI/LXRT with respect to partitioning between different application software modules. Partitioning ensures that a failure caused by a design fault in one application software module cannot propagate to cause a failure in other application software modules, e.g., by blocking access to the CPU or by overwriting memory. Partitioning is important when building mixed-criticality systems comprising both non safety-critical software modules and safety-related ones. Even at the same level of criticality, partitioning improves the robustness of an embedded system. The experimental results described in this paper point out several limitations of RTAI/LXRT Linux concerning fault isolation. Based on these results, we propose modifications to improve the partitioning with respect to temporal and spatial interference.

1. Introduction

The number of Electronic Control Units (ECUs) in cars has increased dramatically in the past twenty years. Today, analysts estimate that more than 80 percent of all automotive innovations are driven or enabled by electronics [1]. Highly distributed systems have evolved based on the logical structuring of the overall automotive electronic system. For the different domains (e.g., powertrain, safety, comfort, multimedia), dedicated clusters with a variety of different communication networks (e.g., CAN, LIN, MOST, Byte-Flight) have emerged. On its behalf, such a cluster consists of a set of ECUs each of which provides a specific function of the domain (e.g., engine control) and lies within responsibility of a single organizational entity (e.g., a particular supplier).

In the past, the dedication of each ECU to a single function has lead to large numbers of ECUs, e.g., up to 75 in luxury cars [2]. The drawbacks of these large numbers of ECUs include the hardware cost and the high numbers of connectors. The latter consequence has increased the likelihood of connector faults, which are a major cause for failures in today's automotive electronic systems [3].

These negative consequences can be tackled by supporting the integration of multiple functions within a single ECU. This strategy is followed by several recent system architectures, such as Automotive Open System Architecture (AUTOSAR) [4], Integrated Modular Avionics (IMA) [5], and DECOS [6].

A key technology for the implementation of such an architecture is an operating system that supports the coexistence of multiple functions within an ECU. This operating system must encapsulate each function in such a way that no interference in the use of the ECU's computational resources (e.g., CPU, memory) can occur. We call such an operating system a *partitioning operating system*. The goal is that the behavior of a function does not depend on the other functions on the same ECU and each function receives guaranteed computational resources as if each function would posses its own ECU. The partitioning operating system needs to prevent interference both in the temporal domain (e.g., start time and duration of availability of a resource) and in the spatial domain (e.g., integrity of code and data).

For a deeper understanding consider an exemplary scenario with two functions. If the two functions share a common ECU with a priority-based operating system, then both functions must be analyzed and understood in order to reason about the correct behavior of any of the two functions. Since the execution of one function can delay the execution of the other functions, arguments concerning the correct temporal behavior must be based on an analysis of both functions.

Missing encapsulation would also blur the organizational responsibilities. If each of the functions operates correctly when executing individually on an ECUs, but exhibit failures after integration when executing collectively on the ECU.

Even faults that are uniquely associated with a function can be difficult to detect, since missing encapsulation would mean that the consequences of a fault can propagate to other functions on the same ECU. For example, a function that overwrites the data or code of another function is likely to lead to an unpredictable behavior of the second function. Even for an external observer, it may be difficult to detect the origin of the fault.

This paper investigates whether a real-time Linux variant can be used as a partitioning operating system. In the automotive domain, Linux is presently seen as suitable for telematics applications (e.g., navigation system, emergency call and location flagging) and for prototyping. Also, adaptations targeting automotive applications have been introduced, such as improvements w.r.t. boot time, temporal performance, and power management [7].

Major advantages of using Linux include the availability of numerous tools and device drivers from both commercial software suppliers and from the open-source community. In addition, Linux can be deployed on the embedded system and also used as the development platform. The benefit is summarized in [8]: *Among other benefits, this provides the ability to build, debug and test applications and libraries on commodity hardware (e.g., standard X86 desktop systems) prior to migration to the development system, resulting in reduced reliance on standard reference platforms and instruction-set simulators.*

In order to evaluate the suitability of Linux RTAI/LXRT as a partitioning operating system, this paper provides experimental results for the error containment of Linux in the temporal and spatial domain. Based on the identified breaches of partitioning, we describe an extension to the Linux variant RTAI/LXRT [9] with improved partitioning capabilities. This extension provides a time-triggered scheduler and a introduces more restrictive Application Programming Interface (API) compared to RTAI/LXRT. The new API eliminates operations that could affect the computational resources of other functions (e.g., no dynamic changing of task priorities).

While this paper focuses on Linux as a partitioning operating system, related work has focused on the development of new real-time operating systems with support for partitioning. For example, solutions for partitioning have been developed in the context of IMA [10]. For example, LynxOS-178 [11] and VxWorks [12] are certifiable operating systems that support time partitioning through a fixed-cyclic time-slice scheduler.

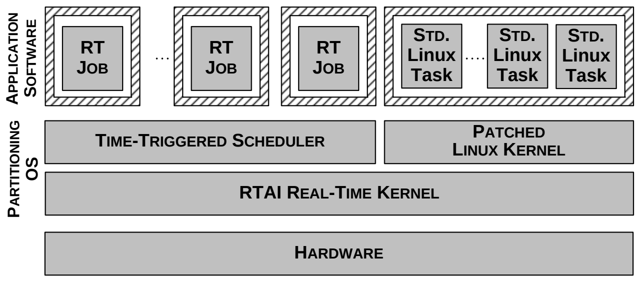

Figure 1. Electronic Control Unit (ECU) Hosting Multiple Encapsulated Jobs

The paper is structured as follows. The focus of Section 2 is the encapsulation of the functions within an ECU by means of temporal and spatial partitioning. The setup and the hypotheses for the experimental evaluation are described in Section 3. Section 4 presents the results w.r.t. the temporal properties and the data integrity when using the introduced partitioning-extension. Also, this section compares our solution with a conventional Linux RTAI/LXRT. The paper finishes with a conclusion in Section 5.

2. ECU with Partitioning Operating System based on Linux

This section explains the realization of an ECU for the integration of multiple jobs. The presented ECU uses a partitioning operating system based on Linux with mechanisms for temporal and spatial partitioning.

2.1. Software Structure of ECU

Figure 1 depicts the constituting elements of an ECU with a partitioning operating system based on Linux. The ECU contains multiple *real-time jobs* each of which provides a part of the overall application functionality. An example of a job in an automotive system would be an *airbag job* that processes values from acceleration sensors and activates the airbags in case of an accident. Such an airbag job must react to the exceeding of acceleration thresholds within a bounded delay (typically 5 ms [13]).

The partitioning operating system must ensure that a job receives guaranteed resources (e.g., CPU, memory), thereby enabling the job to realize its real-time services as in the example of the airbag job. The partitioning operating system in Figure 1 consists of the Linux-kernel, the RTAI real-time kernel, and a time-triggered scheduler.

The *Linux kernel* is patched in order to prevent it from blocking or redirecting hardware interrupts. Hence, Linux cannot add latency to the interrupt response time of the real-time system. RTAI performs these modifications by replacing the corresponding code in the Linux kernel (e.g., cli, sti, and iret statements) with calls to functions in a real-time hardware abstraction layer. This mechanism offers the possibility for software emulation of interrupt control hardware. When an interrupt occurs, the real-time kernel intercepts the interrupt and runs its own dispatcher, which invokes the corresponding real-time handler. In case the interrupt is shared with the conventional Linux kernel, an interrupt pending flag is set. The flag causes the appropriate Linux interrupt handler to be invoked, when the Linux kernel is activated.

Using the Linux kernel, the presented partitioning operating system supports *standard Linux tasks* in parallel with the real-time jobs. Thereby, embedded system developers can leverage existing applications of the open source community. For example, server applications for web services (e.g., http) in conjunction with Linux networking drivers (TCP/IP, Ethernet, etc.) enable maintenance activities via a standard PC with a web browsers. Likewise, Linux tasks permit the establishment of Internet connectivity using available Linux-based networking and security applications (e.g., firewall, encryption).

The *RTAI real-time kernel* provides a real-time scheduler with support for static and a dynamic priority driven scheduling. The real-time scheduler runs the Linux operating system kernel as the idle task by assigning to this task the lowest priority. Hence, non real-time Linux only executes when there are no real-time tasks to run and the real-time kernel is inactive. The scheduling policies supported by the real-time kernel include First In First Out (FIFO), Round Robin (RR), and Early Deadline First (EDF) [14].

In order to prevent temporal fault propagation from the Linux kernel to the RTAI real-time kernel, the real-time kernel is never blocked by the Linux side. For example, the communication links for transferring data between real-time tasks and standard Linux are non-blocking on the real-time side.

On top of the RTAI real-time kernel, the proposed ECU contains a *time-triggered scheduler*. The time-triggered scheduler is a high-priority RTAI task that is executed in kernel space. The time-triggered scheduler performs scheduling decisions at predefined points in time according to a schedule computed by an off-line tool. The benefits of the static temporal control structure created by the off-line tool are the elimination of temporal dependencies between tasks and better predictability and simplicity [15]. Therefore, the presented ECU model and its prototype implementation used in the experiments (cf. Section 3) use a static job schedule that is generated during the system integration phase. As discussed in [16], such a static job schedule can also be used as the starting point for multi-level scheduling. [16] proposes a two-level scheduling algorithm that introduces at lower level a cyclic scheduling and as a second layer priority driven scheduling to schedule computational activities within a static time slot.

A further strength of time-triggered scheduling of tasks is the ability to synchronize computational activities (i.e., task executions) with communication activities (i.e., message exchanges on the underlying network) [17]. End-to-end delays and jitter can be minimized through the temporal alignment of task activations with message reception and transmission instants on a time-triggered network. Task activation times can be set after the reception instants of the time-triggered messages required by the task as input. Likewise, the instant of the task termination (i.e., activation instant plus worst-case execution time) can be aligned with the transmission instant of messages on a time-triggered network, if the task produces output that need to be communicated in messages on the network.

2.2. Spatial Partitioning

Spatial partitioning within a single processor addresses two dimensions: preventing jobs from overwriting memory elements of other jobs (i.e., protection of data and code) and preventing jobs in interfering in the access of devices [18].

In the ECU based on Linux RTAI/LXRT, hardware mediation by the use of memory protection mechanisms is used for establishing spatial partitioning. The processor needs to be equipped with a Memory Management Unit (MMU) [19] distinguishing two modes: supervisor mode (normally reserved to the operating system) and user mode.

In contrast to supervisor mode, memory access is protected by MMU tables in user mode. The MMU tables, managed by the operating system, provide means for spatial partitioning by determining the areas of physical memory that can be accessed by a single job.

In the presented ECU, both standard Linux applications as well as real-time jobs execute in user mode and are protected by the MMU. The real-time jobs utilize the LXRT extension [20] of RTAI, which allows the execution of hard real-time jobs in user mode.

2.3. Temporal Partitioning

In the context of real-time systems, the correctness of a service depends always on two dimensions: the value and the time domain [21]. In an ECU shared among multiple jobs, a primary source for temporal fault propagation is a job delaying other jobs by holding a shared resource (e.g., the processor). The purpose of temporal partitioning is to preserve the temporal properties of resources (e.g., the du-

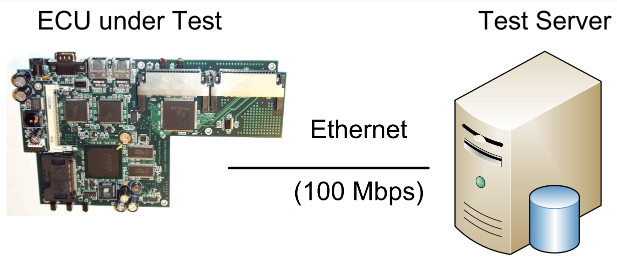

Figure 2. Experimental Framework

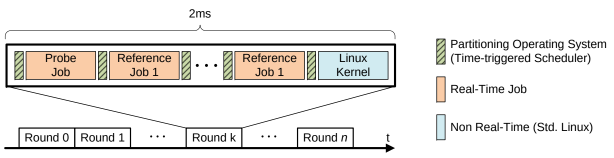

Figure 3. Job Execution in the ECU under Test

ration of availability of CPU or the instant at which the CPU becomes available) even in the case of faulty jobs.

The presented ECU model supports temporal partitioning by design. The time-triggered scheduler activates and preempts jobs at predefined instants regardless of the jobs' behavior. Of course, in an actual implementation of the ECU model the temporal partitioning depends not only on the correctness of the implementation but also on the absence of side channels. An example of a side channel for temporal interference would be a job that initiates a DMA transfer that slows down other jobs.

3. Experiments

This section describes the framework and the experiments used to evaluate the temporal and spatial partitioning of the Linux-based partitioning operating system.

3.1. Framework

The experimental setup is depicted in Figure 2. This setup contains a prototype implementation of the ECU model explained in Section 2 (denoted as ECU under test), and a test server.

ECU Under Test. As the hardware for the ECU under test, we used the Soekris Engineering net4521 embedded system. This compact computer is based on a 133 MHz 486 class ElanSC520 processor from AMD. It has two 10/100 Mbps Ethernet ports, up to 64 Mbytes SDRAM main memory and uses a CompactFlash module for program and data storage. Furthermore, it has two PC-Card/Cardbus adapters which allow an easy extension of the system.

The software configuration used for the prototype implementation combines the ADEOS hardware abstraction layer with a real-time application interface for making Linux suitable for hard real-time applications (we use RTAI v3.1 on a Linux 2.6 Kernel including the real-time RTnet Ethernet driver suite).

Figure 3 depicts the execution of jobs over time. Time is structured into periodically recurring rounds with a duration of 2 ms. In a distributed system with multiple ECUs interconnected by a time-triggered network, these rounds can be synchronized to the time-triggered communication schedule of the network as discussed in Section 2.

In each round a so-called *probe job* is executed which injects specific faults to test the partitioning capabilities of the operating system. Furthermore *reference jobs* are executed the behavior of which is monitored in order to determine whether the faults injected by the probe job have undermined temporal or spatial partitioning. In addition, the non real-time Linux kernel and the Linux-based applications are executed within a time slot.

Monitoring Server. The monitoring server is connected to the ECU under test using an Ethernet connection. The purpose of the monitoring server is to configure the ECU for a specific experiment and to collect monitored data from the ECU. The monitoring server stores this data in a MySQL database for a later analysis.

The monitored data, which is sent back to the monitoring server in order to evaluate the temporal partitioning, includes the following timestamps for each executed job in each round:

- Job activation instant captured by the time-triggered scheduler
- Job activation instant captured by the job (i.e., actual start instant of task execution after context switching overhead)
- Job termination instant captured by the job
- Job termination instant captured by time-triggered scheduler

For spatial partitioning, at the activation instant of a job we calculate four different checksums of read-only memory areas and compare them to the values of the previous round. We compute checksums of the code, the stack, an allocated chunk of heap memory, and an RTAI shared memory.

3.2. Hypotheses

Starting with an analysis of RTAI/LXRT Linux, we have identified scenarios that could lead to violations of temporal and spatial partitioning. We have formulated hypotheses that express the assumed hazards with respect to partitioning. Furthermore, we have formulated hypotheses underlying those mechanisms of Linux that are designed to establish temporal and spatial partitioning. The resulting hypotheses were used as the basis for performing the experiments and the interpretation of the results.

- **Hypothesis 1 – Software and Private Data:** Code, stack and heap memory allocated via the standard Linux system calls is protected by the MMU from being overwritten by any other partition.
- **Hypothesis 2 – Naming Service:** The RTAI naming service has no support for controlling access to registered objects (e.g., a shared memory). A registered object can be accessed by all LXRT tasks that know the name.
- **Hypothesis 3 – RTAI Real-Time Heap Memory.** If a job uses a group heap, the job can overwrite address within the memory region that are in use by another job.
- **Hypothesis 4 – Preemption of Jobs.** Since the time-triggered scheduler has a higher priority than all the partitions it does not matter if a partition tries to block the CPU. It will be preempted anyway and will not affect other partitions.
- **Hypothesis 5 – Priorities.** A partition can set its own priority above the scheduler task of the primary OS, thereby blocking the CPU indefinitely.

3.3. Test Cases

The experiments comprised test cases targeting the MMU, the inter-process communication mechanisms of RTAI Linux, the RTAI memory functions, and the scheduler. During each test case, the probe job injects a specific fault two seconds after the start-up of the ECU.

During the test cases targeting the MMU, the probe job was programmed to access illegal virtual memory addresses (e.g., kernel memory, memory space of another job, NULL pointer). In addition, test cases with an illegal return address on the stack were used to pass control to illegal virtual memory addresses.

In order to evaluate the error propagation paths due to inter-job communication, test cases cover access operation of the probe job to the shared memory of another job. Also, the probe job was programmed to try to free the shared memory of other jobs.

Another group of test cases was targeting the RTAI memory functions. During these tests the probe job modified or closed memory regions of other jobs on the real-time heap.

Finally, we included test cases to determine whether the probe job can affect the CPU time that is available to other jobs. We used a crash failure of the probe job and programmed the probe job to alter its priority, while observing the duration and start instant of availability of the CPU.

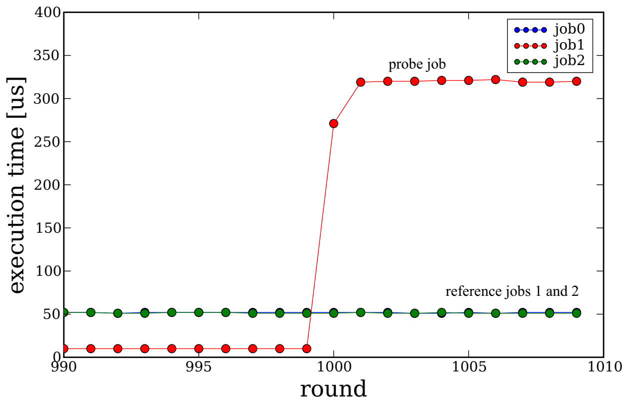

Figure 4. Job Execution Times for Crash Failure of the Probe Job

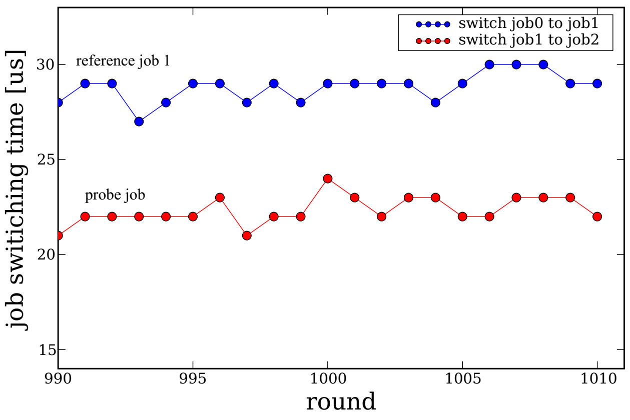

Figure 5. Partition Switching Times for Crash Failure of the Probe Job

4. Results

During the test cases of the MMU (e.g., access to kernel memory, memory space of another job, NULL pointer), no violation of partitioning was observed. Due to the MMU, the probe job exhibited segmentation fault trap and was terminated. Neither was the execution time of the other jobs affected nor was any effect on data integrity observable as indicated by the checksums.

The test cases for the inter-job communication mechanisms and RTAI memory functions, on the other hand, lead to violations of spatial partitioning. The probe job was able to overwrite and free shared memories used for communication between other jobs. In addition, the probe job was able to overwrite the heap memory of another job.

The observed behavior in the test cases for the scheduler depended on the specific fault injected by the probe

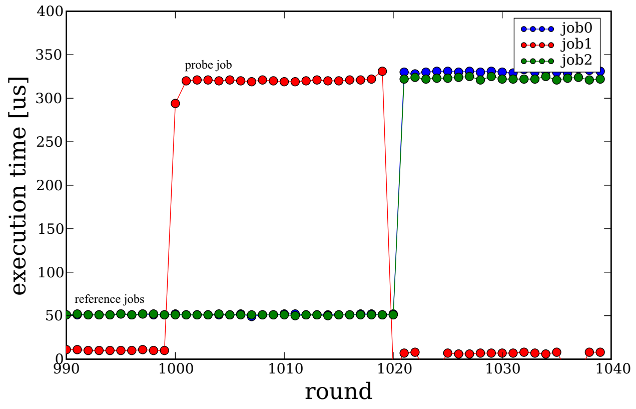

Figure 6. Job Execution Times for Priority Modification Through Probe Job

job. A crash failure of the probe job was handled correctly, i.e., without any effect on the availability of the CPU resources for the other jobs. As depicted in Figures 4 and 5, the probe job is not able to influence either the execution time of the other jobs nor the partition switching times in a significant way. In the testruns during which the crash failure is injected (i.e., after round 999), the execution and switching times of the reference jobs remains unchanged.

The test cases in which the probe job increased its own priority, on the other hand, lead to a crash failure of all jobs on the ECU (see Figure 6). When the probe job set its priority above the priority of the time-triggered scheduler, the probe job was no longer preempted at the instant specified in the time-triggered schedule.

From the point of view of the hypotheses, the anticipated behaviors expressed in the hypotheses occurred. While software and private data are protected by the MMU (hypothesis 1), the naming service (hypothesis 2) and the heap memory (hypothesis 3) permit spatial and temporal interference. Likewise, hypothesis 4 assuming the ability for preempting jobs with crash failures held. Hypothesis 5 also held, which expressed the ability to cause temporal interference through modifying the priority values.

5. Conclusion

This paper has experimentally evaluated the temporal and spatial partitioning of the real-time Linux variant RTAI/LXRT in conjunction with a time-triggered scheduler. The experimental results have uncovered several error propagation paths, which include both temporal interference (e.g., blocking the CPU) and spatial interference (e.g., overwriting memory) between real-time jobs. Consequently, Linux RTAI/LXRT in conjunction with a time-triggered scheduler is not suitable for implementation of embedded systems with application subsystems of different criticality levels.

The major reasons for the error propagation paths are the naming service of Linux RTAI/LXRT and the Application Programming Interfaces (API) provided to real-time applications. Using the naming service, faulty jobs can acquire references to objects of other jobs (e.g., shared memories) and overwrite or free the memory belonging to other jobs.

In addition, the API of Linux RTAI/LXRT provides calls that enable a job to affect the correct operation of the other jobs and the operating system. For example, using an RTAI operating a job can increase its own priority and block the CPU indefinitely.

In order to improve the temporal and spatial partitioning of Linux RTAI/LXRT, modifications are proposed. Firstly, the API should be restricted to a minimum set of operations, which excludes all operations that affect partitioning. Secondly, the naming service should be extended (e.g., using signatures) in order to ensure that only a specific job can acquire a reference to a particular object.

Acknowledgments

This work has been supported in part by the European IST project ARTIST2 under project No. IST-004527 and the European IST project DECOS under project No. IST-511764.

References

[1] G. Leen and D. Heffernan. Expanding automotive electronic systems. *Computer*, 35(1):88–93, January 2002.

[2] A. Deicke. The electrical/electronic diagnostic concept of the new 7 series. In *Convergence Int. Congress & Exposition On Transportation Electronics*, Detroit, MI, USA, October 2002. SAE.

[3] J. Swingler and J.W. McBride. The degradation of road tested automotive connectors. In *Proc. of the 45th IEEE Holm Conference on Electrical Contacts*, pages 146–152, October 1999.

[4] AUTOSAR GbR. *AUTOSAR – Technical Overview V2.0.1*, June 2006.

[5] Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401. *ARINC Specification 651: Design Guide for Integrated Modular Avionics*, November 1991.

[6] R. Obermaisser, P. Peti, B. Huber, and C. El Salloum. DECOS: An integrated time-triggered architecture. *e&i journal (journal of the Austrian professional institution for electrical and information engineering)*, 3:83–95, March 2006. Available at http://www.springerlink.com.

[7] M. O'Donnell. Automotive telematics: Open-source automotive-grade Linux is the future. *Embedded Systems Programming*, July 2005.

[8] D. Beal. A tools-centric view of embedded linux. *EETimes Online*, June 2004.

[9] D. Beal, E. Bianchi, L. Dozio, S. Hughes, P. Mantegazza, and S. Papacharalambous. RTAI: Real-Time Application Interface. *Linux Journal*, April 2000.

[10] Aeronautical Radio, Inc., 2551 Riva Road, Annapolis, Maryland 21401. *ARINC Specification 653: Avionics Application Software Standard Interface, Part 1 - Required Services*, March 2006.

[11] LynuxWorks. *LynxOS 4.0 User's Guide*, 2006.

[12] L. Kinnan, J. Wlad, and P. Rogers. Porting applications to an ARINC 653 compliant IMA platform using VxWorks as an example. In *Proc. of the 23rd Digital Avionics Systems Conference*, volume 2, pages 10.B.1–10.1–8, October 2004.

[13] K. Friedewald. Design methods for adjusting the side airbag sensor and the car body. Technical report, Volkswagen AG, 1998.

[14] L. Dozio and P. Mantegazza. Real time distributed control systems using rtai. In *Proc. of the 6th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*. Politecnico di Milano, 2003.

[15] T.M. Chung and H.G. Dietz. Static scheduling of hard real-time code with instruction-level timing accuracy. In *Proc. of Third Int. Workshop on Real-Time Computing Systems and Applications*, pages 203–211, Seoul, South Korea, October 1996. Sung Kyun Kwan Univ., Suwon.

[16] Y.-H. Lee, D. Kim, M. Younis, and J. Zhou. Scheduling tool and algorithm for integrated modular avionics systems. In *Proc. of the 19th Digital Avionics Systems Conference*, volume 1, pages 1C2/1–1C2/8, Philadelphia, PA, USA, October 2000.

[17] H. Kopetz. Pulsed data streams. In *Proc. of the Working Conference on Distributed and Parallel Embedded Systems (DIPES)*, pages 105–124, Braga, Portugal, October 2006.

[18] J. Rushby. Partitioning for avionics architectures: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center, June 1999. Also to be issued by the FAA.

[19] W. Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, 4th edition, September 2000.

[20] E. Bianchi and L. Dozio. Some experiences in fast hard real-time control in user space with RTAI-LXRT. In *Proc. of the Realtime Linux Workshop*, Orlando, 2000.

[21] H. Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.