Using RTAI/LXRT for Partitioning in a Prototype Implementation of the DECOS Architecture

B. Huber[1], P. Peti[1], R. Obermaisser[1], and C. El Salloum[1]

[1]Institute of Computer Engineering,
Real-Time Systems Group,
Vienna University of Technology, Austria
{huberb,peti,romano,salloum}@vmars.tuwien.ac.at

Abstract — *The steady increase in electronics in automotive systems in order to meet the customers expectation of a cars functionality has led to the development of integrated architectures, as already partly deployed in avionics. Integrated architectures overcome the "1 Function – 1 Electronic Control Unit (ECU)" design philosophy by providing an infrastructure that allows the sharing of ECUs between multiple applications. As a consequence, integrated systems promise massive cost savings through the reduction of resource duplication. In addition, integrated systems permit an optimal interplay of application subsystems, reliability improvements with respect to wiring and connectors, and overcome limitations for spare components and redundancy management.*

In this paper we present an overview of the DECOS integrated architecture and describe a prototype setup. In particular, we describe how inner component partitioning is provided using the open source RTAI/LXRT Linux operating system. We exploit the functionality of this operating system to be suitable for the DECOS architecture and devise a static dispatching technique that forms the basis for the multiplexing of available resources between applications.

1 Introduction

Today, in the automotive domain we find electronic system architectures that cannot be classified as either completely federated nor integrated. Typically, electronic systems in the car do not provide mechanisms to share an ECU among multiple applications and thus do not adhere to the integrated systems design principle. However, for economic reasons, distributed application subsystems share typically the same communication infrastructure (e.g., a Controller Area Network (CAN) for the interconnection of the comfort electronics in a car). This system design has the significant drawback of increased complexity. Since reasoning of the behavior of a particular application requires the understanding of all other applications participating in the communication, this design makes it hard to comprehend the emerging interdependencies between applications. Furthermore, missing error containment capabilities make system integration and diagnosis a very challenging

and cost intensive task that even delay the launch of new car models, since integration responsibilities can hardly be assigned (answering the question who is to blame). In addition, the prevalent "1 Function - 1 ECU" design philosophy leads to a dramatic increase of the numbers of ECUs deployed in today's cars in order to differentiate from competitors and meet the expectations of the customers. For example, luxury cars can have up to 75 ECUs [1]. Since such a high number imposes problems with respect to architecture complexity, wiring, mounting, resource duplication and many others, a reduction of the number of ECUs is of substantial interest.

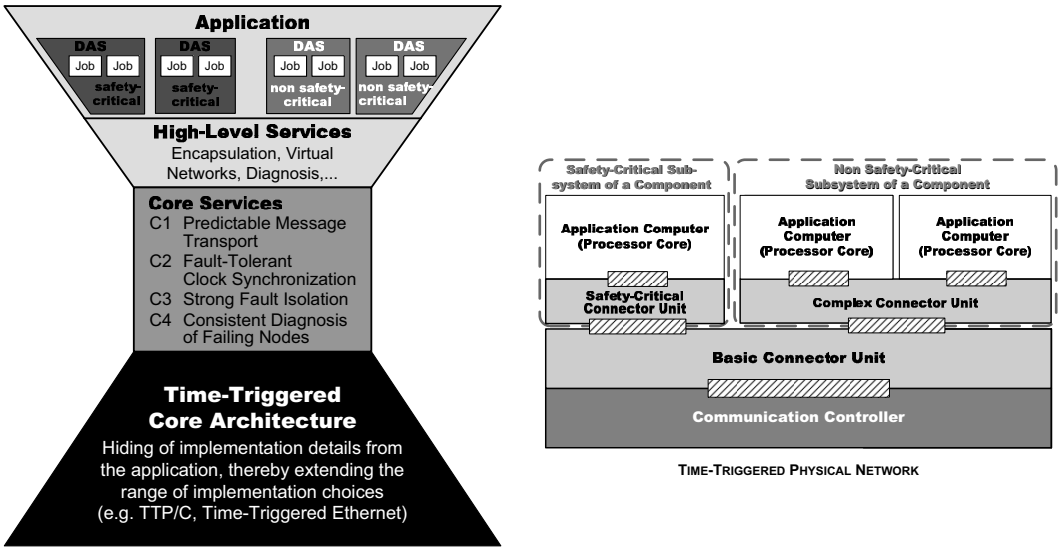
Integrated architectures, such as the DECOS architecture, promise massive cost savings by addressing the problems the automotive industry is currently facing. Via sharing network and component resources among different application subsystems an evolvement beyond the prevalent "1 Function – 1 ECU" strategy is made possible. Integrated architectures not only permit a dramatic reduction in the overall number of ECUs, but also offer increased reliability by minimizing the number of connectors and wires. An ideal future system architecture would thus *combine the complexity management advantages of the federated approach, but would also realize the functional integration and hardware benefits of an integrated system* [2, p. 32].

The DECOS integrated architecture [3] for dependable distributed embedded real-time systems as being developed in the scope of the EU framework 6 program provides a framework with generic architectural services for integrating multiple application subsystems within a single, distributed computer system, while retaining the error containment and complexity management benefits of federated systems. In this paper we present a prototype implementation of the DECOS architecture. In particular, we discuss how spatial and temporal partitioning is achieved in order to provide an environment that allows hosting of multiple application software modules on one component.

The paper is structured as follows. In Section 2 we introduce the DECOS integrated architecture. The prototype implementation's hardware and software setup is described in Section 3. We elaborate on the deployed spatial and temporal partitioning methods of the prototype implementation in Section 4 and in particular describe how the Linux RTAI/LXRT execution environment has been extended to be suitable for the DECOS integrated architecture prototype. In Section 5 we present implementation results such as a timing analysis on the introduced overhead. The paper is concluded in Section 6.

2 DECOS Integrated Architecture

This section describes the DECOS integrated architecture for dependable distributed embedded real-time systems [3]. The DECOS integrated architecture provides a framework with generic architectural services for the integration of multiple application subsystems with different levels of criticality and different requirements within a single, distributed computer system, while retaining the error containment and complexity management benefits of federated systems [2]. Structuring rules guide the designer in the decomposition of the overall system at a functional level and for the transformation to the physical level. In addition, the DECOS integrated architecture aims at offering system designers generic architectural services, which provide a validated stable baseline for the development of distributed applications.



(a) DECOS Integrated System Architecture (b) DECOS Component Model

Figure 1: DECOS System Structure

2.1 Functional System Structure

For the provision of application services at the controlled object interface (e.g., steering actuator), the services of a real-time computer system are divided into a set of nearly-independent subsystems, each providing a part of the computer system's overall functionality. We denote such a subsystem a *Distributed Application Subsystem (DAS)*, since the implementation of the corresponding functionality will most likely involve multiple components that are interconnected by an underlying communication system. The implementation as a distributed system is a prerequisite for establishing fault-tolerance by redundantly performing computations at separate components that fail independently. In addition, the DECOS integrated architecture groups DASs with the same criticality into subsystems (e.g., safety-critical vs. non safety-critical as illustrated in Figure 1).

In analogy to the structuring of the overall system, we further decompose each DAS into smaller units called *jobs*. A *job* is the basic unit of work that employs a *virtual network* [4] for exchanging information with other jobs, thus working towards a collective goal. The access point of a job to the virtual network is denoted as a *port*. Every job has access to its relevant transducers, either directly via the controlled object interface or via a virtual network of known temporal properties.

2.2 Physical System Structure

During the development of an integrated system the functional elements must be mapped to the physical building blocks of the platform. In the DECOS architecture these building blocks are the *time-triggered physical core network*, *components* and *partitions*. The components are part of a distributed computer system and are interconnected by the *time-triggered physical core network*. A *component* is a self-contained computational element with its own hardware (processor, memory, communication interface, and interface to the

controlled object) and software (application programs, operating system) [5]. Components are the target of job allocation and provide encapsulated execution environments denoted as *partitions* for jobs. Each partition prevents temporal interference (e.g., stealing processor time) and spatial interference [6] (e.g., overwriting data structures) between jobs. In the DECOS architecture, a component can host multiple partitions and host jobs that can belong to different DASs.

2.3 Architectural Services

Generic architectural services separate the application functionality from the underlying platform technology in order to enable application software reuse and reduce design complexity. This strategy corresponds to the concept of platform-based design [7], which proposes the introduction of abstraction layers, which facilitate refinements into subsequent abstraction layers in the design flow.

The DECOS architectural services depicted in Figure 1(a) are such an abstraction layer. The specification of the architectural services hides the details of the underlying platform, while providing all information required for ensuring the functional and meta-functional (e.g., dependability, timeliness) requirements in the design of a safety-critical real-time application. The architectural services serve as a validated stable baseline that reduces application development efforts and facilitates reuse, because applications build on an architectural service interface that can be established on top of numerous platform technologies.

In order to maximize the number of platforms and applications that can be covered, the DECOS architectural service interface distinguishes a minimal set of *core services* and an open-ended number of *high-level services* that build on top of the core services. The core services include predictable time-triggered message transport, fault tolerant clock synchronization, strong component-level fault isolation, and consistent diagnosis of failing components through a membership service. The small number of core services eases a thorough validation (e.g., permitting a formal verification), which is crucial for preventing common mode failures as all high-level services and consequently all applications build on the core services. Any architecture that provides these core services can be used as a base architecture [8] for the DECOS integrated distributed architecture. An example of a suitable base architecture is the Time-Triggered Architecture (TTA) [9].

The high-level services of the DECOS architecture include a virtual network service as the encapsulated communication infrastructure tailored to the needs of each DAS [4], an encapsulation service for ensuring inner-component (i.e. job level) error containment, hidden gateways for the interconnection of DAS to improve quality of service and eliminate resource duplication, a redundancy management service (e.g., voting), and a diagnostic service.

2.4 The DECOS Component

In the DECOS component model as depicted in Figure 1(b) we distinguish between two kinds of structuring, *horizontal* and *vertical* structuring. The vertical structuring of the component provides two subsystems within a component. The *safety-critical subsystem* is an encapsulated execution environment for ultra-dependable applications. The *non safety-critical subsystem* offers an environment for those applications having less strin-

gent dependability requirements. For these applications, emphasis lies on low-cost, flexibility, and resource efficiency. The safety-critical and non safety-critical subsystems are established by means of *spatial* and *temporal* inner-component partitioning [6].

As depicted in Figure 1(b) the interconnection between the communication controller and the application computers occurs via *connector units*, which control the application computers access to the state message interface provided by the communication controller. The primary purpose of a connector unit is the allocation of network resources within a component that is vertically structured into two or more subsystems. The connector unit ensures that each subsystem obtains a predefined share of the overall network resources, thus enabling each subsystem to exchange messages with guaranteed temporal properties (maximum latency and latency jitter of message transmissions, minimum bandwidth) and data integrity.

The DECOS architecture does not restrict the choice of implementations of a particular component. Although the provision of separate processors (or processor cores) for each partition has significant advantages with respect to certification, a solution with only one processor shared among the jobs is also an alternative choice as long as the operating system providing the partitioning can be certified up to the highest criticality class [10].

In the DECOS component model we distinguish between three types of connector units (cf. Figure 1(b)). The *Basic Connector Unit (BCU)* performs the primary allocation of the physical network resources, as required for the separation of the safety-critical and non safety-critical subsystems of a component.

The *Saftey-Critical Connector Unit (SCU)* allocates network resources to the jobs of the safety-critical subsystem and realizes the safety-critical high-level services (e.g., voting functionality). In analogy to the BCU, simplicity of the safety-critical connector unit is of major concern in order to control certification efforts.

The *Complex Connector Unit (XCU)* performs the allocation of network resources for the non safety-critical subsystem of a component. Like the SCU, the XCU does not directly access the communication controller, but builds on top of the BCU. This way, the XCU is not involved in the fault isolation and error containment between the safety-critical and non safety-critical subsystem of a component, as this separation is performed by the underlying BCU. Therefore, the XCU and the non safety-critical subsystems of a component need not be certified to the highest criticality levels and the XCU can provide increased functionality at the cost of increased complexity. In contrast to the SCU, the XCU can support the time-triggered and the event-triggered control paradigm. The event-triggered paradigm facilitates the establishment of cost-effective solutions for the non safety-critical subsystems. In this domain, the high flexibility and resource efficiency of event message ports outweigh the lower predictability and increased complexity resulting from the event-triggered control paradigm with its imprecise temporal specifications (e.g., probabilistic queuing models). Furthermore, legacy applications following the event-triggered paradigm can be reused without involving redevelopment efforts. This legacy integration helps in leveraging investments in existing software and lays ground for evolving systems.

For a more detailed description (e.g., constituting hardware elements and software modules) and an analysis of this model with respect to certifyability, encapsulation and independent development aspects refer to [3].

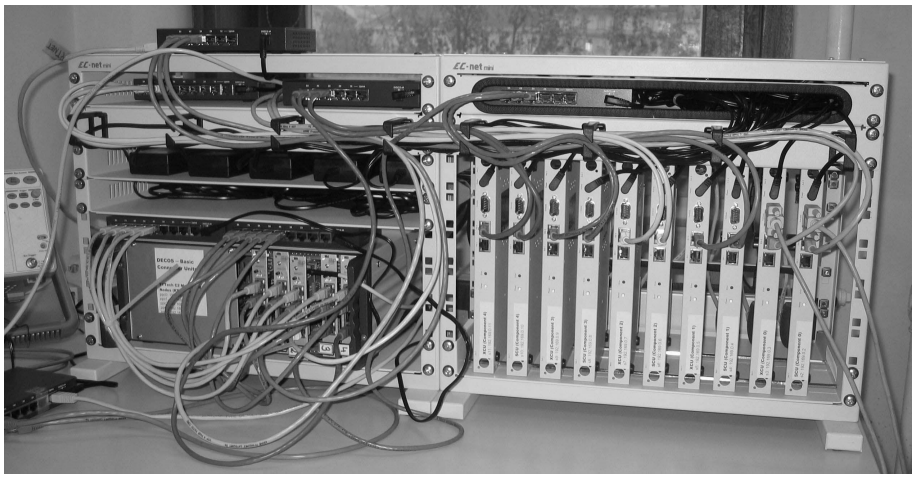


Figure 2: The DECOS Prototype Cluster

3 The DECOS Prototype Implementation

Our prototype implementation of the DECOS architecture consists of a cluster of five components using TTP/C as time-triggered core communication service. Each component hosts a safety-critical and a non safety-critical subsystem with multiple DASs and corresponding jobs. The prototype setup is depicted in Figure 2.

3.1 Hardware Setup

For the components of the prototype cluster we employ distinct hardware elements for the BCU, the safety-critical subsystem, and the non safety-critical subsystem (i.e. one node computer for each SCU/XCU and respective applications). Since we share the same computational resources between the secondary connector unit (i.e. SCU and XCU) and the encapsulated execution environment of the jobs, the need for inner-component error containment by means of temporal and spatial partitioning is raised in order to protect architectural services from job interference on the one hand, and interference between jobs on the other hand.

The interconnection between the BCU and the secondary connector units is realized using time-triggered Ethernet. At a priori defined points in time, an Ethernet message containing the state information that is disseminated on the core network is transferred from the BCU to the secondary connector units and vice versa.

3.1.1 Basic Connector Unit

In our prototype implementation we deploy the TTTech [1] monitoring node [11] as the basic connector unit establishing the time-triggered core communication service (see Figure 3). The TTP monitoring nodes are based on the TTP-C2 controller (AS8202). They are equipped with the Freescale embedded PowerPC processor MPC855T.

In integrated architectures a physical component is shared among multiple jobs. Therefore, it is important to decouple the capability of a node to transmit messages on the

[1] `http://www.tttech.com`

physical time-triggered core network from the functionality of the hosted jobs. In case only a minority of jobs (i.e. software modules) hosted in their respective partitions fail, it still must be possible for other jobs to utilize the architectural services and to communicate via their dedicated virtual network. As a consequence, the communication service provided via the physical time-triggered core network must be decoupled from the application functionality. Note, that according to the DECOS fault hypothesis, the complete node computer is considered to be a Fault Containment Region (FCR) with respect to hardware faults [3].

In the DECOS architecture the decoupling is realized via the connector units. The BCU guarantees message exchange via the core network by not relying on the information provided by the secondary connector units. Such a decoupling is only possible in case a state message interface is provided [12]. A key functionality of the secondary connector units is to map this state message interface of the core network onto the interface required by the jobs of the respective subsystem.

This independence of the time-triggered core communication system from the jobs facilitates a decoupling of the Time Division Multiple Access (TDMA) schedule of the core network from the schedule required by the applications. This means, that the TDMA communication schedule (i.e. the cluster cycle) of the physical time-triggered core network can be significantly shorter than the application requirements in order to provide higher bandwidths. The latency, however, is still limited by the TDMA schedule required by the high level services in order to provide the virtual network service for each DAS.

3.1.2 Secondary Connector Units and Application Computers

For the secondary connector units (i.e. the safety-critical and complex connector units) we use the Soekris Engineering [2] net4521 embedded system as our target computer. This compact computer (depicted in Figure 3) is based on a 133 Mhz 486 class ElanSC520 processor from AMD. It has two 10/100 Mbit ethernet ports, up to 64 Mbyte SDRAM main memory and uses a CompactFlash module for program and data storage. Furthermore, ist has two PC-Card/Cardbus adapters which allow an easy extension of the system.

3.2 Software Setup

Both, the BCU and the secondary connector units use the embedded real-time Linux variant Real-Time Application Interface (RTAI) [13, 14] as their operating system. While in the BCU only kernel modules are used, in the secondary connector units the partitioning capabilities of the RTAI/LXRT extension are heavily utilized.

The software configuration used for the prototype implementation combines the ADEOS [3] hardware abstraction layer with a real-time application interface for making Linux suitable for hard real-time applications (we use RTAI v3 on a Linux 2.4.25 Kernel including the real-time RTnet Ethernet driver suite). RTAI introduces a real-time scheduler, which runs the conventional Linux operating system kernel as the idle task, i.e. non real-time Linux only executes when there are no real-time tasks to run and the real-time kernel is

[2] http://www.soekris.com
[3] http://www.adeos.org

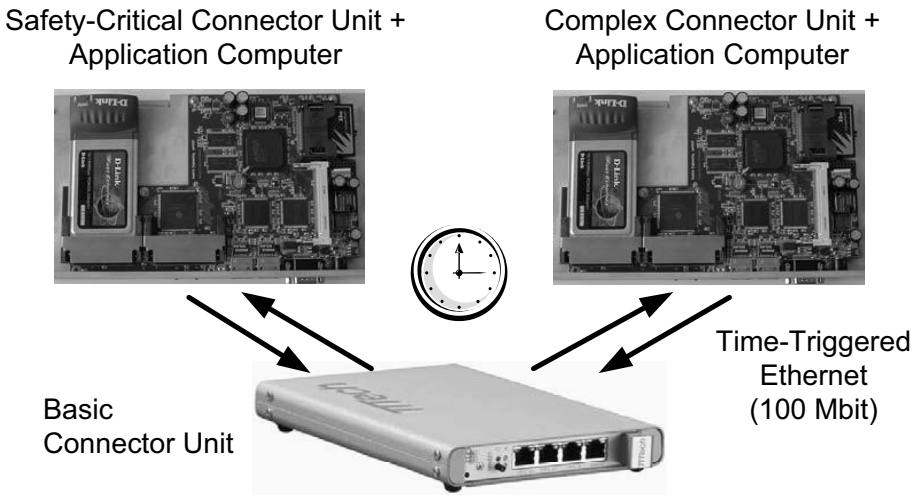


Figure 3: A Prototype DECOS Component

inactive. The conventional Linux kernel is modified 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 with calls to functions in the 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 order to prevent temporal fault propagation from the Linux kernel to the real-time kernel, the real-time kernel is never blocked by the Linux side.

Linux Real-Time (LXRT) is an extension of RTAI that enables the development of hard-real time programs running in user space instead of kernel space. Furthermore, LXRT eases the communication between hard real-time and non real-time processes, which can be utilized for monitoring and debugging of the hard-real time processes, without affecting their real-time behavior.

4 Spatial and Temporal Partitioning Using LXRT

As elaborated in [6], the purpose of partitioning is to ensure that the execution of software in one partition is not affected by software in another partition. Particularly, in case of software failures, the software in one partition must not disrupt the software of any other partition, i.e. partitioning has to inhibit the propagation of software failures between jobs. According to the fault hypothesis of the DECOS integrated architecture, a job is regarded as FCR for software faults [3]. In our prototype implementation, secondary connector units and application computer share the same computational resources. Thus, in case of a software failure of one or more jobs, partitioning is required to ensure the undisturbed execution of the high-level services as well as of the remaining jobs.

4.1 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 [6].

Hardware mediation by the use of memory protection mechanisms is a common way for establishing spatial partitioning. Usually, a processor equipped with a Memory Management Unit (MMU) [15] distinguishes two modes: supervisor mode (normally reserved to the operating system) and user mode. Unlike 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.

Unlike to real-time applications using the RTAI Application Programming Interface (API), which are realized as Linux kernel modules and executed in supervisor mode, real-time applications utilizing the LXRT extension of RTAI retain in user mode. Since LXRT allows the execution of hard real-time jobs in user mode, it preserves the memory protection mechanisms of the Linux operating system and provides support for spatial partitioning. Furthermore, since jobs in DECOS have exclusive access to I/O, explicit synchronization mechanisms for protecting devices are not required.

Communication among jobs is realized via virtual network services, which provide a communication infrastructure tailored to the needs of a particular DAS [4]. Access to these virtual networks is based on messages, using *send* and *receive* operations provided by the architecture. Since the details of implementation are hidden, location transparency is provided to the jobs. In our prototype implementation, high-level services and jobs are hosted on the same computational unit. Thus, shared memory is used for realizing the send and receive operations. In order to prevent an application developer for allocating forbidden memory areas, the required memory areas are determined during the system integration phase and the code sequences for initializing the shared memory are statically linked to the application code.

4.2 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 [12]. In an integrated architecture, a primary source for temporal fault propagation is a job delaying other jobs by holding a shared resource (e.g., the processor). It is the purpose of a scheduler to compile a schedule that fulfills the temporal requirements of all jobs. The purpose of temporal partitioning is the retention of a correct schedule even in the case of faulty jobs holding the shared resource.

Based on the time the scheduling decisions are made, one can distinguish between static and dynamic scheduling. While in static scheduling an offline tool generates a feasible schedule for accessing the shared resources, in dynamic scheduling these decisions are made at runtime. The dynamic scheduling algorithm described in [16] is a preemptive temporal partitioning algorithm that is based on fixed priority scheduling. In order to improve the response time of sporadic tasks, a technique called “slack stealing” is applied that delays the execution of high priority periodic tasks as long as their deadlines are met. The “Proportional Share Resource Allocation” [17] is another dynamic scheduling algorithm that uses discrete-sized time quanta that can be allocated to processes. A weight assigned to every process determines the percentage of the processors capacity the process is to receive.

A static scheduling approach aimed at Integrated Modular Avionics (IMA) systems is

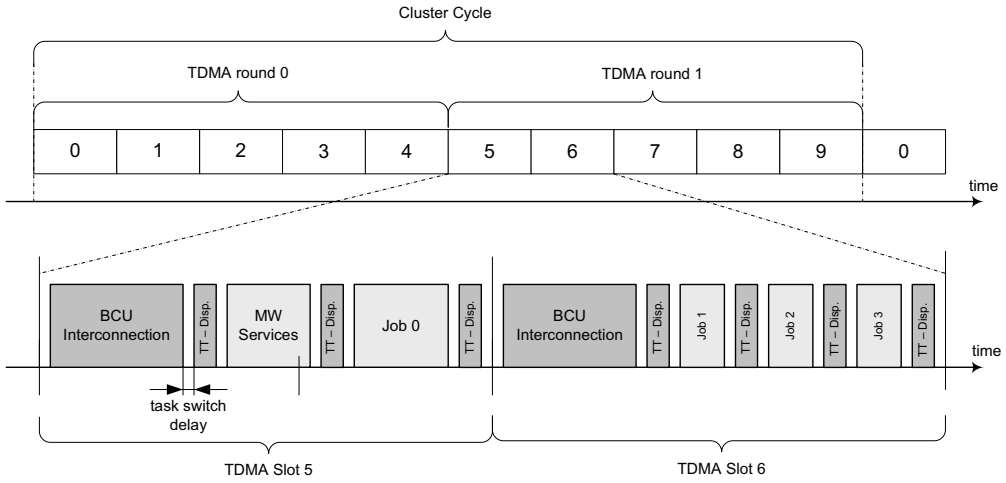


Figure 4: Scheduling Example

described in [18]. A two-level scheduling algorithm is used that provides at lower level a cyclic scheduling to partitions and so-called channel servers which are responsible for message transmission and as a second layer priority driven scheduling to schedule tasks within partitions, respectively messages within channel servers.

While higher flexibility and better support for sporadic tasks are the advantages of dynamic scheduling, predictability and simplicity are the reasons for using static scheduling algorithms. Our prototype implementation depends on a static job schedule that is generated off-line during the system integration phase. Similar considerations as presented in [18] have to be taken into account when developing a feasible schedule. The job schedule is synchronized to the TDMA schedule of the underlying core communication service. It forms the basis of our execution environment and consists of a list of jobs together with their execution times that are to be executed in the respective TDMA slot. Thus, the activation time of a job relative to the starting point of a cluster cycle is given by:

$$t_{job_{n,m}} = n \cdot t_{slot\_length} + t_{comm} + m \cdot t_{dispatch\_overhead} + \sum_{i=0}^{m-1} t_{exec\_job_{n,i}} \qquad (1)$$

With n as the slot position in the TDMA schedule (starting with 0), t_{slot_length} the length of one TDMA slot, t_{comm} the time required for the inter-connection of BCU and secondary connector units, m the job position within slot n, $t_{dispatch\_overhead}$ the scheduling and dispatching overhead, and $t_{exec\_job_{n,i}}$ the reserved execution time of job i in the slot n. A sample schedule is shown in Figure 4.

In order to guarantee temporal partitioning it has to be ensured that jobs are not able to exceed their assigned processor time in the case of a software fault of a job. Thus, assuming the correctness of the static schedule, temporal partitioning is assured, if a job is activated and has finished (or is deactivated in case of failure) within its associated time-slot. RTAI offers mechanisms for periodical time-triggered scheduling of real-time processes, but it lacks of the possibility to specify a deadline or a maximum execution time until a real-time process has to have finished its execution. Therefore, we have developed a time-triggered dispatcher extending the RTAI/LXRT functionality that activates

```
Initialize_dispatcher_task();
Obtaining_LXRT_task_references();

Wait_for_timed_activation();

while (!end_of_task_list) {
  rt_task_resume(task_list[i]);
  rt_sleep(task_exec_time[i]);
  rt_task_suspend(task_list[i]);
  i++;
 }
```

Figure 5: Pseudo Code of the Time-Triggered Dispatcher

high-level services and jobs according to a static timetable and ensures that their assigned execution times are not exceeded. Figure 5 describes the function of the dispatcher using pseudo code. All jobs that are to be executed on the secondary connector unit (this includes also the high-level services) are LXRT hard real-time jobs, i.e. they are executed in user space. At node startup they are loaded and initially suspended. Loading the jobs means that they use the RTAI *naming and registering* service to register a name to a pointer to their task structure in RTAI. This name can be used by all RTAI/LXRT jobs to obtain a reference to the job.
The time-triggered dispatcher is realized as an RTAI Linux kernel module. After loading and initializing the module, an RTAI task running in kernel space with highest system priority is created. This task analyzes the static schedule and obtains a reference to all LXRT jobs of its scope. After this initialization, the dispatcher releases the processor and waits for re-activation by calling the blocking *rt_sem_wait()* system call. Since the task schedule is synchronized to the TDMA schedule of the interconnection between BCU and secondary connector unit, the receive interrupt of the RTnet driver module realizing the time-triggered Ethernet interconnection is used to activate the dispatching of LXRT jobs. Therefore, the RTnet driver has been adapted to trigger the dispatcher after a complete time-triggered Ethernet message has been received. The actual dispatching is done in three steps:

1. According to the static schedule, the respective LXRT real-time job is activated by using the RTAI call *rt_task_resume(task_ptr)*. *task_ptr* is added to the *Ready-Queue* of the RTAI scheduler, but the job is not executed due to its lower priority.
2. The dispatcher temporary releases the processor by calling *rt_sleep(job_exec_time)*. Internally, RTAI moves the dispatcher task to the *Delayed-Queue* and hands over the processor to the previously activated LXRT job. *job_exec_time* denotes the amount of time that is granted to the job.
3. After the specified amount of time has elapsed, RTAI activates the dispatcher task again. Due to the higher priority of the dispatcher, the LXRT job is preempted and suspended by using *rt_task_suspend(task_ptr)*.

Due to this enforced preemption by RTAI and the dispatcher task, the LXRT real-time jobs do not need to act cooperatively and release the processor. Thus, it is prevented by

our execution environment that a faulty job delays other jobs by holding the processor longer than specified.

5 Discussion and Results

Using RTAI/LXRT combined with the time-triggered dispatcher of our prototype implementation, we provide a simple but powerful framework for executing several jobs and architectural services on the same computational unit. Due to the use of Linux as the operating system, device drivers, standard tools and a multitude of programming languages can be used for application development.

Intellectual Property (IP) Protection The protection of IP is a major requirement of automotive suppliers and vendors in order to maintain their competitive edge. The encapsulation of jobs in dedicated partitions supports this IP protection by providing the possibility to supply jobs as pre-compiled object files, thus rendering obsolete the need for supplying source code. In our prototype implementation, this property is realized by using an LXRT wrapper code that is linked to the object file of the application and takes care of the correct initialization and instantiation of the LXRT real-time task as well as other required resources (e.g., allocation of shared memory regions) and the correct activation of the job. Therefore, the application code has to provide an initialization routine called *init_point()*, which is called once by the wrapper after setting up the LXRT environment. The periodic functionality of the job is accessed by the *entry_point()* routine, which is called by the wrapper each time it gets activated by the dispatcher.

Evaluation of timing constraints In our prototype implementation, the time-triggered dispatcher is replicated on all secondary connector units and thus, the static task schedules executed on the individual nodes are independent of each other. The number of tasks (including jobs and architectural services) executed on a secondary connector units is limited by the TDMA slot length of the BCU to SCU/XCU interconnection and the overhead introduced by the time-triggered dispatcher. The minimum time required for TDMA slot j is described by the following equation:

$$t_{slot_{j,min}} \geq t_{comm} + n_j \cdot (t_{dispatch} + t_{tswd} + t_{job_{j,n}}) \tag{2}$$

t_{comm} denotes the time required for the interconnection of the BCU and the secondary connector units. It includes the time for receiving the time-triggered Ethernet frame disseminated from the BCU as well as for assembling and transmitting a new Ethernet frame back. In our current implementation using the *RTnet* driver module, t_{comm} equals to 380 μs. In order to reduce jitter on task execution, a 10 μs buffer for jitter compensation is added to t_{comm}. The number of jobs that are to be executed in slot j are denoted by n_j. The overall overhead introduced by the time-triggered dispatcher, i.e. task selection, task activation, and task suspension is covered by $t_{dispatch}$. This overhead equals to 18 μs. t_{tswd} covers the overall task switch delay introduced by RTAI. This delay is not constant, but jitters between 40 μs and 65 μs in our current implementation. The granted processor time of job n in slot j is denoted by $t_{job_{j,n}}$.
It is important to note that the task switch delay time t_{tswd} occurs only once in equation 2.

The processor is granted to a job by calling $rt\_sleep(t_{job_{j,n}})$ in the dispatcher. Thus, RTAI ensures that the dispatcher is activated again in time, and the execution time of the job is reduced by the additionally needed task switch delay.

System Integration System integration unites the separately developed DASs into the overall system (e.g., the car). Currently manufacturers are facing substantial problems during system integration. An integrated system approach must provide solutions that reduce integration time and efforts (and consequently reduce integration costs). In combination with a diagnostic architecture, the encapsulation services provided via the use of RTAI/LXRT and the time-triggered dispatcher allow tracing for faulty jobs (i.e. software faults). This enables to precisely assign *integration responsibilities* to both, the system integrator as well as the vendors that have independently developed jobs or even a complete DAS to be integrated into the final system. The proposed architecture precludes vendors from transferring integration responsibilities for their jobs to the system integrator, since the IP protection mechanisms (integrator cannot modify code) in combination with the encapsulation services (both at network and component level) allows tracing integration problems back to the vendor of the respective job or DAS.

6 Conclusion and Future Work

The DECOS integrated architecture provides an infrastructure that facilitates the deployment of both, safety-critical and non safety-critical Distributed Application Subsystems (DASs) in a single distributed computer system. The presented prototype cluster on the basis of the open source RTAI/LXRT Linux operating system implements a partitioning strategy that allows multiplexing of resources between jobs hosted on one physical component. The introduced partitioning strategy realizes also important properties of the DECOS architecture such as Intellectual Property (IP) protection and job location transparency. In addition, the provision of encapsulated partitions in combination with a diagnostic subsystem enables the assignment of system integration responsibilities, which is a fundamental prerequisite to enable independent development of jobs or complete DAS by different suppliers.

First promising experiments have shown that the employed partitioning strategy prevents jobs from disturbing the execution of other jobs hosted on the same node. However, further investigations by means of software fault-injection experiments are required in order to validate the resilience of the presented encapsulated execution environment. Furthermore, optimization of the BCU to secondary connector unit interconnection is required to reduce the minimal TDMA slot length at application level and thus the latency of the Distributed Application Subsystems (DASs).

Acknowledgments

This work has been supported by the European IST project DECOS under project No. IST-511764.

References

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

[2] R. Hammett. Flight-critical distributed systems: design considerations [avionics]. *IEEE Aerospace and Electronic Systems Magazine*, 18(6):30–36, June 2003.

[3] H. Kopetz, R. Obermaisser, P. Peti, and N. Suri. From a federated to an integrated architecture for dependable embedded real-time systems. Technical Report 22, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-1, 1040 Vienna, Austria, 2004.

[4] R. Obermaisser, P. Peti, and H. Kopetz. Virtual networks in an integrated time-triggered architecture. In *Proceedings of the Tenth IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS2005)*, February 2005.

[5] H. Kopetz and N. Suri. Compositional design of RT systems: A conceptual basis for specification of linking interfaces. In *Proceedings of Sixth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, pages 51–60, May 2003.

[6] 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.

[7] A. Sangiovanni-Vincentelli. Defining platform-based design. *EEDesign of EETimes*, February 2002.

[8] J. Rushby. A comparison of bus architectures for safety-critical embedded systems. Technical report, Computer Science Laboratory, SRI International, September 2001.

[9] H. Kopetz and G. Bauer. The time-triggered architecture. *IEEE Special Issue on Modeling and Design of Embedded Software*, January 2003.

[10] Radio Technical Commission for Aeronautics, Inc. (RTCA), Washington, DC. *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*, December 1992.

[11] TTTech Computertechnik AG. TTP Monitoringnode - A TTP Development Board for the Time-Triggered Architecture, March 2002.

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

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

[14] RTAI Programming Guide, Version 1.0. Dipartimento di Ingegneria Aerospaziale Politecnico di Milano (DIAPM), Italy, September 2000. Available at `http://www.rtai.org`.

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

[16] P. Binns. A robust high-performance time partitioning algorithm: The digital engine operating system. In *Proceedings of 20th Conference on Digital Avionics Systems*, volume 1, Daytona Beach, FL,USA, October 2001. Honeywell Laboratories, Minneapolis.

[17] I. Stoica, H. Abdel-Wahab, K. Jeffay, S.K. Baruah, J.E. Gehrke, and C.G. 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.

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