An Extensible Partitioning Framework for Safety-Critical Systems

Felix Bruns*, Dirk Kuschnerus*, Anas Showk* and Attila Bilgic†

*Ruhr-Universität-Bochum
44801 Bochum, Germany
{felix.bruns, dirk.kuschnerus, anas.showk}@ruhr-uni-bochum.de

†KROHNE Messtechnik GmbH
47058 Duisburg, Germany
a.bilgic@krohne.com

Abstract—Certification of safety-critical systems requires a separation of certified and non-safe components. Up to today, partitioning operating systems, that have the capability to isolate software components in safety-critical systems, are almost exclusively found in complex embedded systems with standardized runtime environments, e.g. in the avionic industry. On the other hand, their use is highly uncommon in deeply-embedded systems, that are frequently characterized by severe power, memory, and latency constraints. Here we show, how partitioning can be efficiently provided in deeply-embedded systems. For this purpose, we propose a framework for highly customized and constrained embedded devices, which achieves the separation of software components using a legacy real-time operating system. The approach focuses on the flexibility and low complexity of the framework in order to minimize the effort for safety certification. The framework is modular and extensible.

Index Terms—Safety-critical, RTOS, partitioning, modular, extensible

I. INTRODUCTION

Software for safety-critical systems is subject to strict regulation concerning its development and verification [1]. In order to minimize development and certification effort, software developers aim to isolate and minimize safety-critical parts and reuse already certified software whenever possible. However, in order to prevent a faulty application from compromising the systems integrity, a strict separation between certified, dependable software and non-safe software has to be maintained. Especially in the avionic industry, numerous real-time operating systems (RTOSs) with the ability to partition software have been developed. This means, they have the ability to isolate applications of different criticality and certification into separate domains [2]. Thereby they can serve as a platform for redundant or fault-tolerant software execution. Furthermore, they offer a stable interface to applications and therefore support the modularization and reuse of software.

Another alternative to create a partitioned system is to use virtualization technology. In this approach, a hypervisor emulates a virtual execution environment for one or several real-time operating systems and prevents interference between them. Thus, virtualization enables the reuse of legacy operating systems. In contrast to avionic computer systems, highly embedded devices usually consist of small, optimized micro-control-units (MCUs) that do not provide a high computing performance and advanced features such as memory management units (MMU) or hardware support for virtualization. Therefore, embedded devices for safety-critical applications usually maintain a physical isolation of software components by executing them in separate MCUs. In addition, embedded software is optimized for a specific application and frequently is dependent on peripheral devices. For this reason, RTOSs for embedded applications only provide minimal functionality such as synchronization, inter-process communication (IPC) and timekeeping. Device drivers on the and system services on the other hand, are usually developed by the device vendor. The diversity in hardware and software, as well as the high level of optimization and specialization found in embedded software, severely complicates the development of software partitioning mechanisms and reusable software modules.

Despite the problems mentioned above, software partitioning can be beneficial for systems with considerable power or size constraints, which are common e.g. in industrial measurement systems [3] or ambulant medical applications. A software partitioned system can be smaller than a hardware partitioned solution, since all the software is executed in a single MCU. In addition, it promises lower power consumption, because no bus communication between the various domains is necessary and because no synchronization is required for the power management of several MCUs.

In this work, we examine software partitioning for deeply-embedded systems with severe power constraints. We show how spatial partitioning can be provided on top of legacy RTOSs. For this purpose, we developed a partitioning framework which allows device vendors to reuse customized or inhouse-developed RTOSs, device drivers and services. Consequently, the framework allows the reuse of a certified or proven code base with minimal modification. Changes to certified code require an evaluation and re-certification and are therefore costly. In order to minimize the certification effort in a diverse and changing runtime environment, the framework adopts a module-based approach and can be easily extended for new functionality without modifying and re-certifying core components. We demonstrate our approach using the open-source FreeRTOS [4] operating system. Similarly to our approach, virtualization also enables the reuse of legacy RTOSs. We discuss benefits and drawbacks of both approaches and analyze their impact for deeply-embedded systems.
This work is organized as follows: In the next section we give a brief review related work. In Section III we discuss the requirements for temporal and spatial partitioning before outlining the realization in Section IV. Section V introduces a common, generic API for service invocation. In Section VI, we carry out an abstract evaluation of our approach and compare it to a virtualized system, and finally conclude this work in Section VII.

II. RELATED WORK

Partitioning OSs are frequently used in avionic systems and have been standardized in [2]. Similarly, the AUTOSAR-standard [5] describes a component-based OS architecture and infrastructure for safety-critical automotive systems. Both standards are aimed at significantly larger and more powerful computing systems than our solution. In particular, they require a standardized runtime environment, whereas our partitioning framework is intended for diverse and customized systems which are prevalent in most embedded markets.

Our solution shares several aspects with the microkernel approach for secure and reliable systems, such as the focus on a minimal code base. In particular, the IPC method and access control scheme is inspired by recent L4 microkernel implementations [6]. Furthermore, the modular concept is similar to the Coyotos microkernel [7] which is described as "extensible object system" and the theory of access control in extensible systems is described in [8]. Several systems exist to allow the reuse of an existing RTOS in a safety-critical context, e.g. the Xtratum-hypervisor [9] which is close to the ARINC standard for avionic systems. Another example is the SPIRIT-microkernel [10] which allows to execute existing RTOSs on top of a virtualization layer. In contrast to these systems, the partitioning framework allows to wrap an existing RTOS in a safety layer, and thereby achieve an even smaller safety-critical code base. Furthermore, due to the modular structure, a hybrid approach is possible that allows to place performance-critical functionality inside the operating system kernel without modification and re-certification of other operating system parts.

A configurable, scalable system architecture is common in many RTOS, e.g., in eCos [11], but these system do not take certification effort and the isolation of domains into account.

III. REQUIREMENTS

Partitioning of applications can be decomposed into spatial and temporal partitioning. Spatial partitioning means that applications cannot modify the data of other applications in an unwanted way. In a straightforward implementation, spatial partitioning requires some form of hardware assisted memory protection, such as a memory protection unit (MPU) or MMU. However, direct protection of memory alone is not sufficient when it can be bypassed by manipulating OS functionality. Consider a fairly standard system call in an RTOSs:

\[ \textit{enqueue(struct queue_t* targetQueue, void* srcBuff)} \]

By giving incorrect value for \textit{targetQueue}, a faulting application can overwrite critical operating system data or peripheral registers and thereby modify or crash the whole system. Worse, it could reference a legal queue structure of a different domain and thereby provide faulty input data to a safety-critical application, or block the buffer space in such a way that the safety-critical application can no longer communicate reliably. Consequently, system calls have to be extended for access control and argument validation in order to contain errors in the faulting domain.

Temporal partitioning resorts to the elimination of timing dependencies between applications. Thereby it is assured, that safety-critical applications always receive enough processor time to fulfill their functionality. Using a standard RTOS kernel, full temporal partitioning cannot be provided, since a special scheduling scheme, such as time-triggered scheduling, is required. Using priority-based scheduling, it is possible to give safety-critical applications higher priority than less critical functionality and thereby order applications according to their integrity. If no blocking dependencies, such as mutexes, are shared between the applications, a low integrity application cannot influence higher priority applications. Another possibility is to simply use round-robin scheduling. In both cases timing dependencies between domains may exist, but are not dangerous since sufficient processor utilization and latency bounds can be guaranteed. Timing dependencies can however have negative effects on predictability. Finally, it is possible to extend existing RTOSs with hierarchical scheduling [12].

Temporal partitioning is not mandatory in fail-safe systems which can enter a safe mode in case a failure is detected. In these systems it is usually preferable to detect a timing error, e.g., with a watchdog, and then to enter the fail-safe mode. Then a safe system can be guaranteed at the cost of a reduced reliability.

IV. PARTITIONING LAYER

The purpose of the partitioning framework is to provide spatial partitioning in the form of memory protection, access control and argument validation for a legacy RTOS while minimizing certification effort. To achieve these goals we target a minimal code size and clear internal interfaces to avoid code modification and re-certification. Although not strictly necessary for safety-critical applications, the framework also prevents memory reads and other information transfers if not necessary for the functionality of the system. This improves fault detection and could enable a deployment for security-related purposes. Moreover, system services such as fault handlers, loggers and device drivers can be registered in the framework and it thereby serves as center for a module-based, extensible design. The framework supports ARM Cortex-M3 processors with MPU.

A. System call interface

The OSs code and data regions are protected by the MPU against disallowed access by applications. Therefore a well-defined entry point has to be provided to transfer control to the partitioning OS. This is in contrast to an unprotected RTOS,
were system calls are identical to function calls. The transfer of control is realized with a software interrupt (SWI), which also causes a change to a privileged processor mode. System call arguments are transferred to the operating system kernel in the data registers of the processor. Since application and RTOS are not compiled together, this has the additional benefit that OS and applications can be modified independently of each other. If a non-preemptive host RTOS is used, the system call can simply be handled in the context of the software interrupt. However, to enable preemptive task switching during an ongoing system call, the system call must be executed in the context of a protected, privileged thread as shown in Figure 1. Therefore a dedicated stack memory for kernel threads is required in order to use preemptive functionality. The software interrupt forwards system call arguments to the kernel stack and dispatches the privileged kernel thread. After the system call has been handled, the calling thread has to be rescheduled and the return values have to be forwarded. The drawbacks of a preemptive system are additional memory consumption, due to the privileged kernel stacks and a significant run-time overhead caused by mode switches. Consequently, system calls are significantly more costly and should be used less frequently than in a non-preemptive or unprotected RTOS.

The dispatching of privileged handler threads can be bypassed with a compiler switch and therefore the partitioning framework can be configured for use with preemptive as well as non-preemptive RTOSs. In this work we applied the preemptive mode since some system calls in FreeRTOS require preemption to function correctly.

B. Access Control

Access control in the partitioning framework is regulated by domains. Each thread in the system belongs to a domain, which allows access to certain memory regions and OS functionality. Furthermore, a fault handling strategy can be registered in the domain.

In order to regulate system calls, each domain locally holds a list of proxy objects [13] [14]. A kernel object, such as a task or message buffer, can only be made visible to a domain by creation of a local proxy. Compared to a global access control list, this has the benefit that objects can be referenced by a local ID or name and consequently different domains are not restricted by a shared namespace. This access scheme is closely related to capability-based access control in secure OSs, e.g., in [7]. However, secure systems have to delegate rights in an open and dynamic environment, whereas proxies in the partitioning framework cannot be created or modified after the initialization of the system.

The proxy object further restricts access by allowing only a pre-configured subset of operations, e.g., only read-accesses on a message buffer. Effectively, a proxy is a placeholder for a kernel object of arbitrary type and, if access is allowed, forwards system call arguments which have to be decoded and interpreted by the underlying module. For this purpose, modules have to implement a common kobject-interface (shown in Listing 1). This access scheme makes it possible to add new types and system calls to the kernel, without modifying the core of the partitioning framework. Consequently modifications and effort for re-certification can be contained to the modified module.

The resulting access scheme is depicted in Figure 2. The functionality of FreeRTOS is integrated in the framework by adding wrapper modules, which validate systemcall arguments and then forward the system calls to the host RTOS. In addition to FreeRTOS functionality, stand-alone modules, e.g., for application level interrupt control, are provided. Each module implements the common interface kobject and can therefore be integrated without modification of the partitioning framework. It is however necessary to provide customized functions to encode the system calls on the application level.

C. Inter process communication

Shared services or resources such as peripheral devices are in the ideal case encapsulated in their own domain. The resulting design is more robust, since errors are contained, and more modular, because dependencies must be resolved through the OS interface. On the other hand, a high amount

---

Listing 1: Kobject interface - invoke is a system call with generic arguments

```c
error_t invoke(Kobject_t* kobj, void* arg0, void* arg1, void* arg2, syscall_t callId);
```

---

Fig. 1: Mode changes during system calls in a preemptive kernel

---

Fig. 2: UML diagram of the module-based access control scheme. Black boxes show independent optional modules.

### TABLE 1: Overhead of the partitioning layer for service requests

<table>
<thead>
<tr>
<th></th>
<th>execution time (normalized)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FreeRTOS queue</td>
<td>100</td>
</tr>
<tr>
<td>P.F. queue</td>
<td>179</td>
</tr>
<tr>
<td>P.F. message gate</td>
<td>137</td>
</tr>
</tbody>
</table>

---
of messages have to be passed between domains and hence an efficient IPC mechanism is required. In short, the resulting system bears strong resemblance to microkernel-based systems and shares many of the advantages and shortcomings. Additional difficulties are posed by the partitioning and safety requirements. For instance different domains must not share a common message buffer to prevent resource conflicts. Furthermore, dynamic allocation of memory as well as complex synchronization schemes such as priority inheritance are difficult to realize in a safe way.

To resolve some of these problem, a dedicated buffer type for inter-domain communication called message gate was developed using the FreeRTOS queue data type. It is dangerous to share Mutexes, semaphores and messages queues across domains due to the risk of run-time dependencies or service denial. These dangers can be avoided by the use of message gates. A message gate is a queue buffer for incoming messages which must be connected to another message gate in a different domain. Only point-to-point connection between message gates is allowed. Consequently, each connection has its own dedicated buffers and resource conflicts between different domains are impossible. The connection is explicitly registered in the OS during the initialization of the system and the behavior such as blocking- or non-blocking operation is configured. It can therefore be checked if the communication patterns between server and clients are consistent, and, if blocking communication is used, that no run-time dependencies on uncerified applications exist.

Usually a service is requested by sending a message and waiting for a reply. Using the standard communication facilities of an RTOS, this message transfer is particularly inefficient. As can be seen in Figure 3(a), four separate system calls have to be executed to handle a request and hence, the overhead of kernel entry and exit accumulates repeatedly. The overhead can be reduced by a dedicated send-and-wait system call for service requests, which directly waits for a reply after sending the message (Figure 3(b)). As can be seen in Table I the overhead of service requests can be reduced from 79% to 37% compared to native FreeRTOS system calls. This solution still does not achieve the full efficiency of message passing in microkernel systems, since no direct copy from sender to receiver domain occurs. It provides however a safe and efficient way of communication with minimal additional complexity.

D. IRQs

To minimize the amount of safety-critical code, hardware-related software, such as device drivers, should be preferably executed as applications in a protected domain. Then, provided that the hardware device that the driver controls cannot compromise system safety, a faulty driver cannot compromise the safety of the system as a whole. In order to achieve functional device drivers on the application level, it is necessary to provide them with a way to control the hardware device, and the interrupts that the device triggers. Access to the registers of the peripheral device is granted to a specific domain by configuration of the MPU. However, a special kernel module is required to control and receive interrupts. Using the IRQ-module, it is possible to use the proxy mechanism to give control of a specific IRQ to a domain. This scheme is shown in Figure 4. The domain can then use systemcalls to activate or mask the interrupt. To receive an interrupt, a thread can use a blocking wait on a semaphore or a similar kernel mechanism. Similar to the registering of proxy objects, it is possible to register a interrupt handle in an IRQ handler table. The interrupt handler table uses the message interface described in Section V to transmit an IRQ event. Since it uses a fixed interface, it is independent of the specific mechanism and can be reused with different OSs and mechanisms.

A faulty peripheral device could create a high interrupt load or a permanent IRQ. To prevent an overload of the system, the interrupt is masked by the operating system and remains pending until it can be cleared by the application. Therefore the operating system scheduler can control the load caused by interrupt processing.

E. Exceptions

Exceptions, such as memory protection faults, usage faults etc. are trapped in the kernel. The task which caused the
Listing 2: To achieve a fixed interface for message passing the kobject interface can be extended

```c
error_t send(Kobject_t* kobj, void* sndBuff, size_t sndSz);
error_t receive(Kobject_t* kobj, void* rcvBuff, size_t maxSz, size_t* rcvdSzOut);
error_t sendAndWait(Kobject_t* kobj, void* sndBuff, size_t sndSz, void* rcvBuff, size_t rcvMaxSz, size_t* rcvdSzOut);
```

Table II: Benefits and Drawbacks of different service implementations

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Spatial partitioning</th>
<th>Temporal partitioning</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread (time triggered)</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>Thread (preempting)</td>
<td>+</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>OS Kernel</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

Table II: Benefits and Drawbacks of different service implementations

A specific system service, e.g., a device driver, can be provided by a user level server or as functionality in the OS kernel. Table II gives an overview of different implementation options. If a service is implemented as user level thread it is possible to contain memory errors and thereby the certification effort is reduced. However, the exchange of messages results in an increased overhead. In addition, low latency service requests are only possible if the server has a higher priority than the caller which leads to a violation of temporal partitioning. Then it can be difficult to prove that a system is free of priority inversion and meets its deadlines. These problems are avoided if the service is implemented as a part of the kernel, since it is invoked in the temporal context of the caller. On the downside, because the service is executed in the privileged processor mode, it can influence the whole system and therefore must be certified according to the highest level of integrity. Another problem occurs in distributed systems when a service is requested that is executed on only one MCU. Then the service has to be requested differently depending if the caller is on a remote MCU or on the same MCU. As can be seen, the constraints of the specific system determine the most appropriate implementation choice for the runtime environment and influence the way a service has to be requested.

Different implementation choices for a service should not influence applications. Otherwise, the application has to be modified and re-certified if it is ported to another system. Similarly, the way that notifications such as interrupts or exceptions are signaled to user level should not directly depend on services by the host RTOS. It should be possible to reuse the partitioning framework and runtime environment with different host RTOSs without extensive modifications.

In order to enable a reuse of applications and services with a different host RTOS, we extended the kobject interface with an option for service invocations (Listing 2). The extended interface provides a fixed semantic for sending and receiving signals and service request, which can be easily implemented by different modules. As a result, it can be statically configured which module provides a specific service and therefore a flexible configuration of the runtime environment is possible. A use case for the flexible implementation of services is shown in Figure 5. By configuration of the proxy it can be determined if the request is forwarded to a user level task, handled internally or even transmitted to a remote MCU. A drawback of this approach is a lack of type safety since different kinds of services can be requested with the same syscall. Therefore, the message interface checks if the underlying service realizes a specific protocol, e.g., logging service, time service, etc. An
VI. COMPARISON TO HYPERVISOR SOLUTIONS

Several alternatives exist to reuse an existing RTOS in a partitioned system. A legacy RTOS can be reused on top of a hypervisor to achieve a virtual execution environment for the system. Processor supported full-virtualization could, in principle, enable to reuse the legacy RTOS unmodified, but is currently only supported in high performance, application CPUs. Since we focus on severely power- and performance constrained systems, para-virtualization, where an operating system is adapted to use an API provided by the hypervisor instead of directly controlling the processor, seems to be the most feasible alternative, due to its low overhead and independence of processor support.

For an abstract comparison of the partitioning framework and a para-virtual solution, we consider an unprotected “library OS” such as FreeRTOS, which includes an inter-process communication mechanism, and a mechanism for bus communication for use in distributed systems. In order to achieve a safe, partitioned operating system using para-virtualization, it is necessary to modify the low level mechanisms which control the hardware and adapt them to the programming interface of the hypervisor. This approach is shown in Figure 6. In the illustrated case, this would include a virtualization of the bus communication in addition to the virtualization of the CPU, since a single hardware bus has to be shared between two virtual systems and consequently the hypervisor has to regulate access to the shared resource.

Using the partitioning framework on the other hand, the core components of the RTOSs including low level code (e.g. thread switching, bus drivers, etc.) can be reused unmodified, but high level wrappers and control code have to be added in order to guarantee the isolation of domains. In addition it is necessary to adapt applications to the new programming interface. The modification of applications is trivial - only the type of the first argument changes from a pointer type to an object Id, but the systemcalls themself have to be encoded since they are transmitted by a SWI.

Comparing the two architectures for performance impact, certification effort, and flexibility shows that both have certain advantages.

- **Spatial partitioning:** Both architectures are capable of full spatial partitioning.
- **Temporal partitioning, latencies and efficiency:** Temporal partitioning in hypervisor and partitioning kernel, is usually realized by time-triggered, static scheduling scheme and non-blocking communication between domains[15]. Consequently, the processing of events, e.g. interrupts or messages, has to be executed in a polling mode. The additional overhead and processing delay caused by periodic polling suggest that such a system is not well suited for deeply-embedded systems that require a high efficiency and short processing latencies. In contrast to temporal partitioning OSs, the RTOSs commonly applied for deeply-embedded systems provide efficient, low latency scheduling and communication mechanisms. However, due to the lack of temporal partitioning, the field of application of the partitioning framework is limited to fail-safe systems. In addition, the domains cannot be developed and tested independently of another due to possible temporal interferences. This is probably a serious drawback in the development process of complex systems.

- **Memory requirements:** Since the hypervisor requires additional memory, and the legacy RTOS runs duplicated in both domains, a significantly larger memory footprint will be required in a virtualized system than in the partitioning framework.

- **Performance:** Both architectures will have certain runtime overhead compared to an unprotected RTOS. Using the partitioning framework, many systemcalls, e.g. for IPC, will require additional en- and decoding as well as argument checking. On the other hand, using a virtualized architecture, a significant overhead will occur each time the legacy RTOS requires support by the hypervisor, e.g. for thread switching, interrupt handling or bus communication [16]. However, certain functionality of the legacy RTOS such as shared memory communication can be executed with very low or even without additional overhead [17]. It will therefore depend on the specific application which architecture choice provides the best performance. We expect, that a para-virtualized architecture will provide a lower performance for most deeply-embedded systems, since these systems are frequently
characterized by a high amount of interrupt handling and short thread execution times.

- **Certification effort:** We expect a significantly higher certification effort for the virtualized architecture, since a complete set of operating system mechanisms such as scheduling, IPC, access control, etc. have to be developed to provide the functionality of the hypervisor. Furthermore, the changes in the legacy RTOS to achieve para-virtualization can be intrusive and error-prone. Practically they are similar to porting the legacy RTOS to a different processor architecture. Using the partitioning framework on the other hand, most operating system mechanisms can be reused unmodified. Due to the extensible design, shared peripheral drivers such as the bus communication in Figure 7 or additional system services can usually be included unmodified in the partitioning OS. In a minimal system, only a system call interface, access control and memory protection mechanism have to be added.

- **Flexibility:** The virtualization approach provides a higher flexibility than a partitioning OS. Once a certified hypervisor is available, several guest OSs can be virtualized and separated from each other with moderate effort.

With the partitioning framework, a spatially partitioned system can be realized with relatively low additional certification effort and will usually be more economical than the development of a hypervisor. Furthermore, the system should provide a higher level of efficiency than a solution using full temporal isolation or virtualization technology. However, adding performance optimizations to adapt the operating system mechanisms to a partitioned environment and mechanisms to improve temporal isolation increase the effort. It is possible to choose if operating system services are provided inside the operating system kernel or as protected mechanisms to a partitioned environment and mechanisms to improve temporal isolation increase the effort. It is possible to choose if operating system services are provided inside the operating system kernel or as protected functionality on the application level. Therefore it is possible to optimize a specific system for performance or safety. This choice is helpful for embedded systems with severe constraints, but can be a burden in highly complex applications where many system services have to be provided and can interact in unforeseen ways. In theses systems a fixed, standardized set of operating system abstractions, as provided by a hypervisor or microkernel, is preferable to reduce the complexity. Furthermore, a hypervisor can be used to provide partitioning on several, distinct legacy RTOSs and is therefore the better solution in systems were different operating systems have to be supported.

**VII. Conclusion**

Our analysis showed that the reuse of a legacy RTOS in a partitioned OS is a feasible alternative for deeply-embedded systems, especially if the system has severe latency or power constraints. The system should however be fail-safe, because only limited temporal partitioning can be achieved with most RTOSs. Using the partitioning framework, a spatial separation of software components was achieved with the help of the industry proven FreeRTOS kernel. Furthermore, we created an efficient and safe communication mechanism between domains with a communication layer on top of FreeRTOSs IPC mechanism. In addition, we described an extensible access control scheme and abstractions for service invocations which ensure that our approach is not limited to FreeRTOS and can be easily adapted to different operating systems. By use of the framework, a spatially partitioned system can be achieved with low effort and moderate additional complexity. Further optimizations were shown, that improve the performance and safety in systems with microkernel structure. However, these optimizations increase the complexity without achieving the full performance of dedicated microkernel. Consequently our approach is best suited for low complexity systems, where system services can be included in the operating system kernel or do not have to be shared between domains.

**References**


