Virtualization for Safety-Critical, Deeply-Embedded Devices[*]

Felix Bruns
Integrated Systems
Ruhr-University Bochum
Bochum, Germany
felix.bruns@rub.de

Dirk Kuschnerus
Electronic Circuits
Ruhr-University Bochum
Bochum, Germany
dirk.kuschnerus@est.rub.de

Attila Bilgic
KROHNE Messtechnik GmbH
Duisburg, Germany
a.bilgic@krohne.com

ABSTRACT

Even today, safety-critical systems in many fields of application use separate processors to isolate software of different criticality from another. The resulting system architecture is non-optimal in regard to flexibility, device size and power consumption. These drawbacks can be prevented by the use of partitioning operating systems that enable the integration of applications with different criticality on a single processor. However, their application for deeply-embedded devices, that are characterized by strict resource constraints and the lack of advanced processor features such as memory-management units (MMU), is challenging. In this work, we show that the impact of virtualization on performance and predictability is smaller in the field of deeply-embedded devices than in more complex systems, making it a compelling choice as a partitioning technology. We present a hypervisor that provides time and space partitioning for an MMU-less system, as well as mechanisms for communication and resource sharing. To satisfy the strict power and resource constraints found in deeply-embedded devices, we focus on solutions with a minimal runtime overhead. Furthermore, the hypervisor is integrated with the processor power management, often enabling significant power savings in the resulting system architecture.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection; D.4.8 [**Operating Systems**]: Performance

1. INTRODUCTION

In safety-critical systems, a separation of safety-critical and non-critical software is mandatory. Pioneered in the avionic industry [17], partitioning operating systems provide a space and time separation between software of differing criticalities or certification status. Faults are contained in the defective partition by memory protection, access control and a static, time-triggered scheduling scheme. Since applications in different partitions cannot influence each other in disallowed ways, they can be developed and certified independently of another. As a consequence, the process of safety certification is significantly simplified. Furthermore, partitioning operating systems provide a flexible system architecture which enables an optimization for safety concerns.

However, in most industries with safety constraints, the introduction of partitioning operating systems has not gained traction. The underlying reasons are strong resource constraints such as power consumption, memory requirements and processing capacity.

The adoption of partitioning operating systems (OSs) is especially difficult in severely power-constrained systems. This is the case for industrial measurement and control devices that have to operate in hazardous atmospheres [6], as well as for ambulant medical devices or wireless sensors. Due to the strong power constraints found in these industries, computations are performed by small micro-control units (MCUs). These systems have only a few kilobytes of RAM at their disposal and do not contain a memory management unit (MMU) or advanced virtualization features. Usually a small real-time operating system (RTOS) or even a bare-metal application serves as runtime executive. We refer to this class of systems as *deeply-embedded*.

In deeply-embedded systems, the partitioning of applications of different criticality is performed by a physical separation of the subsystems on different MCUs. This approach is called the federated architecture. Hence, a deeply-embedded device for a safety-critical application usually features several MCUs, which are interconnected by bus communication.

Whereas the federated architecture has good isolation qualities, it results in a strongly limited flexibility. The system designer is bound by a hard limit on the number of partitions, and has few options to optimize the software architecture towards safety or security goals. Furthermore, physical separation is non-optimal in terms of device size and power consumption, since several separate MCUs must be supplied and communication is performed by costly bus transactions. Related to the problem of safety, security challenges emerge due an increasing interconnectivity and the expansion of computing devices into new fields of application. More flexible architectures are required to handle the

[*]©2013 ACM. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in Proceedings of the 2013 ACM Symposium on Applied Computing, March 2013

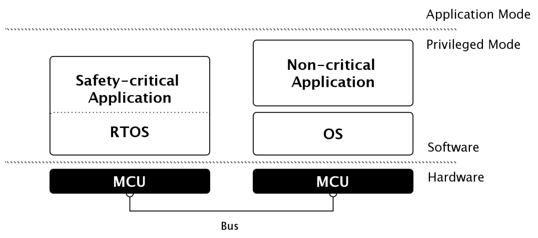

Figure 1: Federated architecture of safety-critical systems. Safety-critical and non-critical subsystems are physically separated on distinct MCUs.

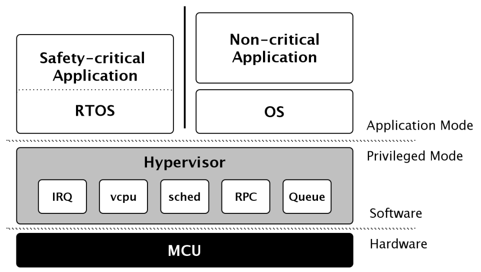

Figure 2: Use of a partitioning hypervisor to create an integrated architecture.

conflicting safety and security issues. All of these problems question the suitability of the federated architecture for future device generations.

The improved flexibility provided by an integrated architecture enables the development of enhanced safety concepts and security architectures and leads to improvements in software certification and reusability [14]. As outlined above, an integrated architecture can also lead to a reduced power consumption and smaller devices. To exploit these benefits, the partitioning OS has to achieve a high predictability and minimal resource consumption in order to satisfy the constraints for deeply-embedded devices. Furthermore, it must be highly efficient, because otherwise the additional runtime overhead reduces or even exceeds the power reduction that can be accomplished by abolishing physical separation.

In this work, we argue that the problem of partitioning under severe resource constraints can be solved by applying virtualization techniques. We show, that for deeply-embedded systems, the overhead of virtualization is minimal due to three factors: Firstly, a bare-metal system or a system using a simple RTOS can be virtualized with minimal hypervisor interaction. Secondly, modern MCUs for deeply-embedded markets are optimized for low IRQ latencies, and therefore enable to handle hypercalls and the delivery of exceptions very efficiently. And finally, MCUs for deeply-embedded applications usually do not have cache memories and therefore do not suffer from increased cache contention due to the intervention of the hypervisor.

In addition, virtualization can be of remarkable importance for safety-critical systems, since it allows to reuse safety-certified operating systems and applications in a new environment, significantly reducing the need for costly and longsome recertification.

In this work, we present a hypervisor which provides space and time partitioning for deeply-embedded applications. To our knowledge, we present the first hypervisor for deeply-embedded applications and demonstrate its performance. In addition, we compare the power consumption of a virtualized system to that of a federated approach. Furthermore, we present communication mechanisms that allow to decrease the runtime overhead compared to traditional solutions and still maintain a temporal separation between safety-critical and non-critical subsystems.

2. CONCEPTS AND TECHNIQUES

2.1 Architecture

A diagram of a federated architecture which can frequently be found in safety-critical systems is shown in Figure 1. Safety-critical and non-critical subsystem are executed on two distinct MCUs. Sometimes, a complex operating system such as Linux can be found in the non-critical subsystem, in order to maximize reusability and ease user interaction. But usually, a *libOS* is used, that means the applications are linked to the underlying RTOS. Both, application and RTOS run in the privileged processor mode. In this setup, a systemcall is equivalent to execution of a function and therefore the overhead of systemcalls is minimized. As we will show later, this property allows for a very low virtualization overhead [16]. The only way of interaction between the MCUs is a bus interface, which is used to transfer data or request services on the other partition. Bus communication is relatively slow, costs a significant amount of energy and creates additional runtime overhead for protocol processing and checksum calculation.

Figure 2 shows the architecture that we applied in our system. In order to achieve a separation on a single processor, it is necessary to move the safety-critical and non-critical system to the deprivileged application mode. Otherwise, fault containment cannot be achieved. The partitioning kernel operates in the privileged processor mode. It assures space and time partitioning and provides basic system services to the partitions. However, transferring control between the privileged processor mode and the application mode is costly. In order to achieve high efficiency, the number of mode switches has to be minimized. This can be achieved by virtualization. Since it results in a certain autonomy of the partitions from the underlying partitioning kernel, most functionality can be handled by the guest OS internally, without the need for costly processor mode switches.

The processors used in deeply-embedded systems do not offer virtualization support. However, virtualization can still be achieved with a para-virtualization approach. Using para-virtualization, hardware dependencies in the guest OS are replaced by hypervisor functionality. As a result, para-virtualization is independent on virtualization features of the processor and therefore applicable in deeply-embedded systems. Furthermore, it has only a small penalty on performance and, since the manufacturer usually has control of the device's source-code, is relatively easy to achieve.

To achieve a para-virtualized system, we developed a small research hypervisor, optimized for use in deeply-embedded systems. It provides space partitioning between the subsystems by a static access control scheme and use of the processors memory protection unit (MPU). Since virtual memory is not provided by the MCU, applications have to be linked to specific address regions. Time partitioning is provided by a cyclic scheduling scheme. Although the hypervisor is non-preemptive, IRQs and partition switches are not sig-

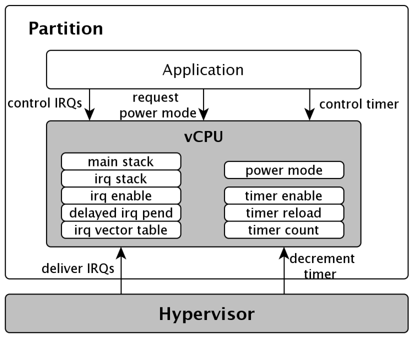

Figure 3: A virtual CPU structure is the main interface to the hypervisor. It emulates a physical processor.

nificantly delayed since all hypercalls are short (below 500 cycles in the worst-case). In our setup, the hypervisor runs on ARM Cortex-M3 MCUs and, in a configuration with two partitions and several communication channels, adds only about 7 kB ROM and 1 kB RAM to an existing system.

2.2 Virtual processor interface

We achieve para-virtualization by use of a virtual processor interface (vCPU). The use of virtual CPUs has been shown to be more efficient than other approaches for para-virtualization [10]. Since it provides a very similar interface to a real processor, the guest ported to the vCPU abstraction with fewer modifications, resulting in a lower runtime overhead and porting effort.

The structure of a vCPU is shown in Figure 3. The vCPU resides in the partition's address region and is freely accessible to the partition. Most interaction between the guest OS and the hypervisor can be performed using the vCPU interface. Therefore it is seldomly necessary to invoke a costly hypercall. An intervention of the hypervisor is only required for the delivery of IRQs, partition scheduling and inter-partition communication. In contrast, communication inside a partition, synchronization between threads and scheduling can be handled autonomously by the guest OS, resulting in a very low virtualization overhead.

The main task of the hypervisor is IRQ delivery, which is done in FIFO order. If an IRQ is pending for a specific vCPU, the hypervisor stores the program position and writes the partition's current stack pointer to the *main stack* variable of the vCPU. Then it modifies the stack pointer and program counter of the partition. The partition is resumed using the *irq stack* at the program position of the corresponding entry in the *irq vector table*. When returning from an IRQ, the procedure is performed in reverse.

Real-time operating systems usually synchronize by frequent use of critical sections. For a highly efficient virtualization, it is therefore mandatory that critical sections can be used without resorting to costly hypercalls. Using a virtual processor as interface, the guest OS simply sets or clears the *irq enable* bit in the vCPU. In case an IRQ for the specific partition gets pending, the hypervisor checks if the partition is in a critical section. If the partition is in a preemptable state, the hypervisor delivers the IRQ. Otherwise it sets the *delayed irq pending* bit. The partition has to check the *delayed irq pending* bit when exiting a critical section and, if necessary, process a delayed interrupt.

In contrast to a more complex system, the guest OS is agnostic of memory protection or memory management features in a deeply-embedded device. Consequently, this functionality does not have to be provided by the hypervisor. Furthermore, the guest OS usually consists of a library that is linked with the application, whereas system calls are issued by software interrupts in a general-purpose system. Therefore the hypervisor usually does not have to forward system calls. As a result, the virtualization overhead for deeply-embedded operating systems and software is significantly smaller than that of complex, general-purpose OSs.

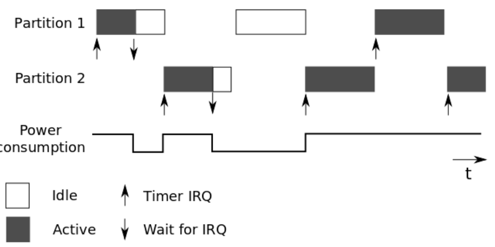

Figure 4: A schedule of a system with two partitions. Partition 1 requests a period of two timeslices and enters the idle mode. Partition 2 requests a period of one timeslice.

2.3 Partition Scheduling

Partitions are scheduled by the hypervisor in a simple cyclic scheme. Each partition receives the same amount of processing time and is scheduled with the same frequency. The cyclic scheme realizes a reservation based policy in a very simple way. This policy was chosen, because it provides a very low scheduling overhead. Therefore, the length of the timeslices can be made very small in order to enable short IRQ latencies. Since the guest OSs realize their own scheduling policy on top of the hypervisor, the complete system has a hierarchical scheduling policy [18].

The partition scheduler is furthermore responsible for providing a timebase to the partitions. For this purpose, it realizes a virtual timer peripheral that is controlled by the partition using the timer registers in the vCPU interface. The scheduler keeps track of the timer's state. If it has expired, it reloads the timer and delivers a timer interrupt to the partition. The timer can for example be used by partitions as scheduling timer or as watchdog.

An exemplary schedule for a system with two partitions is shown in Figure 4. The partitions control the virtual timer peripheral and cause the hypervisor to deliver timer IRQs at fixed multiples of the activation times. If a partition is idle, it can voluntarily issue a *waitForIrq()* hypercall. As a consequence, the partition yields control of the processor as long as it does not have pending IRQs, enabling the hypervisor to use the processors low-power mode.

2.4 Low-power Policy

Frequently, power-efficiency is the most critical constraint in deeply-embedded devices, and it is therefore important that the partitioning approach does not lead to an increased power consumption.

The opportunities for voltage frequency scaling are substantially limited in deeply-embedded systems, since many peripheral devices require a fixed voltage level and because providing several voltage levels requires additional energy. Furthermore, the error rates of memories, processors and peripheral devices rise strongly when the supply voltage is decreased [21][7], making voltage scaling in safety-critical applications risky.

As a result, most systems apply a race-to-halt policy, that means they execute at a high clock frequency and enter a low-power mode as fast as possible. In this way, the static power consumption can be reduced, often below the level that can be achieved by voltage frequency scaling [3].

Most MCUs provide several low-power modes. In a standard low-power mode, which we call *idle mode*, the processor core is deactivated, but still connected to the power line. All peripheral devices remain functional, leading to power savings of approximately 75%. In a *deep-sleep* mode, the MCU and most of the peripheral devices are disconnected from the power line and hence do not consume static nor dynamic power. In this case, the power consumption is strongly reduced, often to the low μW range. Deep-sleep modes can however only be entered in special circumstances, when the application is not depending on peripheral devices.

To make efficient use of the processor's low-power modes, the hypervisor synchronizes with the partitions to choose a low power policy. Partitions can request a specific low-power mode in a register of the vCPU. When a partition enters the wait for IRQ mode, the hypervisor checks the state and the requested low power mode of all partitions. If all partitions are in a wait for IRQ state and agree to deep-sleep, the hypervisor can safely enter the deep-sleep mode. The system has to be woken by an IRQ of one of the few peripheral devices that remain active in the deep-sleep mode. When an IRQ arrives, the hypervisor is woken, resumes normal operation and delivers the IRQ to the respective partition.

The effectiveness of the power mode synchronization is increased by a *timeslice donation* policy. When a partition requests to wait for interrupts, but other partitions remain active, the scheduler cannot enter the deep-sleep mode, since the active partitions might be relying on peripheral devices. That means, that only the idle mode can be used. Instead of using the relatively ineffective idle mode, the scheduler searches for another partition that can be scheduled for the remaining timeslice. As a result, excess processing time is transferred between partitions, enabling the scheduler to bundle active intervals. Consequently, frequent switching between idle mode and active mode is avoided and the system can enter a deep-sleep mode as soon as possible. On the next timeslice after a donation, the scheduler returns to the cyclic scheduling scheme, guaranteeing a fixed minimum progress for each partition.

Timeslice donation can be of special benefit in systems with mixed timing constraints. The policy allows to transfer execution time slack [12] between the partitions. In most devices, tasks without real-time constraints or with soft constraints can be found in addition to hard, safety-critical workloads. Using timeslice donation, it is possible to speculate that execution time slack received from other partitions helps to complete these tasks. As a result, the total processing power in the system can be reduced, leading to additional power savings.

An exemplary schedule under the use of the low power policies is shown in Figure 5. When one of the partitions enters the idle mode, its processing time is donated to the other partition, enabling it to finish without switching between power modes. When both partitions are waiting for interrupts, the scheduler checks the requested power modes of both partitions, and enters the appropriate low-power mode.

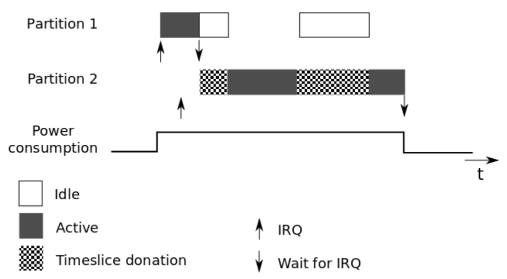

Figure 5: Example for timeslice donation. When Partition 1 is idle, it donates its processing time to Partition 2, enabling it to finish earlier. Timeslice donation reduces the number of processor mode switches and improves the use of deep-sleep modes.

3. COMMUNICATION MECHANISMS

When a partitioning OS is used to execute applications of mixed criticality on a single MCU, some services and devices usually need to be shared between the partitions. This can for example be achieved by providing a system partition, which is responsible for shared devices and provides services to the other partitions. As a result, there can be a significant number of communication connections and messaging between the partitions, requiring efficient and safe communication services.

In avionic systems, communication between partitions is provided by point-to-point connections [2]. Each communication connection is an exclusive resource and therefore resource contention between partitions cannot occur. Receiving from a point-to-point connection can only be done in a polling mode in order to avoid temporal interferences between the partitions. The resulting system has good separation qualities, but a significant runtime overhead, since a high amount of unnecessary polling operations takes place. Furthermore, the resulting communication latencies can be very long, since messages have to traverse the static, time-triggered scheduling scheme.

In order to minimize the overhead for inter-partition communication, we provide two communication mechanisms. An asynchronous queue mechanism, which is well-suited for communication of applications with differing criticality, and a Remote Procedure Call (RPC) mechanism which allows to efficiently invoke a service on a trusted partition and therefore is optimal to realize shared services. The communication connections and the required buffers are created and registered in the hypervisor during the startup phase of the system. For this purpose, the hypervisor uses a configuration that is specified by the system's designer. Both services are protected by the use of an access control scheme. Only those partitions that are explicitly allowed to use a specific communication connection can do so.

3.1 Message Queues

Figure 6 shows a diagram of the queue mechanism, which enables asynchronous communication. It provides two modifications over the traditional approach used in avionic systems. Instead of providing a point-to-point connection, several senders can send to a single receive port. In this way, the

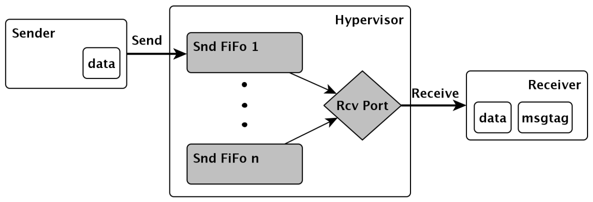

Figure 6: Asynchronous communication is provided by multiple-sender-single-receiver connections. Since each sender has a dedicated buffer, the isolation between partitions is guaranteed.

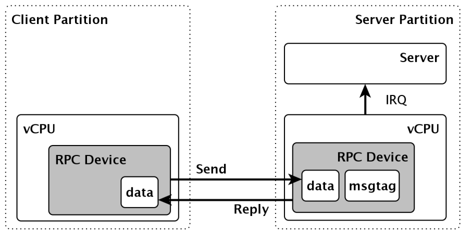

Figure 7: Remote procedure calls are performed using a special communication device which is part of the virtual CPU. The vCPU triggers an interrupt to inform the application of an RPC request.

receiver can avoid the repeated overhead of checking several connections. Resource contention between the senders cannot take place, because each sender has a dedicated FIFO-buffer. The receiver can choose between receiving from the last sender, or using round-robin scheduling between active senders. In addition to the message, a message tag is transferred to the receiver, which holds information about the sender and the received message. In this way the receiver can identify the sender and check if it has the rights to invoke the requested operation.
Furthermore, a partition can enter the idle mode until a message is available on one of its receive ports, leading to a reduction of unnecessary polling operations and lower power consumption.

3.2 Remote Procedure Calls

Remote Procedure Calls allow to invoke a service that is shared with other partitions. The sender of an RPC operation is referred to as client, and the receiver as server. The client is blocked for the duration of the RPC and continues processing using the received answer once the RPC has finished. In the ideal case, an RPC operation simply looks like a function call.

When a RPC operation is requested by a partition, the hypervisor performs an instant switch to the server partition. That means, that the service is provided by another partition using the timeslice of the client. Therefore, the server partition does not have disadvantages when providing a service and cannot be overloaded by faulty clients.

In order to embed the RPC functionality in the virtualization interface, we implemented the receiver side of a RPC as a virtual device. The RPC device contains a data buffer, as well as several fields for message identification. Each partition can have several RPC devices. When a RPC message is sent to a server, the data is transferred directly from a buffer in the client to a local buffer in the server and an IRQ is triggered. The RPC is then handled in a special interrupt service routine. When the server has finished processing, an answer is returned to the clients buffer and control is returned to the client.

If the server is handling another IRQ or a critical section, the server schedules the RPC just like a normal IRQ. It continues processing in its current state and processes the RPC directly when leaving the non-preemptable state. This arrangement allows the server partition to handle a request as fast as possible and still synchronize with other functionality by the use of critical sections. It is not necessary for the server to perform a blocking call on a receive routine or periodically poll on a receive port. As a result, the abstraction of the server as an independent virtual processor is sustained.

The RPC operation must be finished before the timeslice of the requesting partition ends. Otherwise, the shared resource is blocked and might only be available to other partitions after a delay, leading to a reduced timing predictability and a possible denial-of-service situation.

In order to avoid that the timeslice runs out during a RPC operation, a worst-case-execution-time (WCET) for the RPC connection is registered during the systems configuration. When a partition requests an RPC, the hypervisor compares the remaining length of its timeslice to the registered WCET. If the RPC can be performed during the timeslice, it is permitted. Otherwise, the hypervisor stores a continuation structure [8] for the client partition and enters the idle mode for the rest of the timeslice. The RPC is continued the next time the client is scheduled.

If a timeslice runs out during an ongoing RPC, this might indicate a serious error in the server. The hypervisor interrupts the ongoing RPC, triggers an exception for the server partition and resumes scheduling normally. In the client, the RPC operation returns with an error value. In this way, the participants of the RPC operation are aware of the error and can react if the safety of the system is compromised.

In effect, the RPC mechanism allows to use shared services in a time-multiplexed way. Client partitions can act as if they had exclusive access to the resource. Therefore the use of a RPC mechanism also allows smaller delays and results in a better timing predictability than the use of asynchronous communication.

4. EVALUATION

We test the performance and the power consumption of our hypervisor on a low-power MCU running at 14 MHz and 28 MHz respectively. This MCU is designed for use in deeply-embedded systems. It has no caches, only a few kilobytes of RAM and and a few tens of Flash memory. It is however optimized towards low IRQ latencies. When an exception occurs, the processor automatically stores a part of the context on the stack and restores it on exception exit, resulting in a total overhead of only 24 cycles. This feature enables to switch between the application and the hypervisor rapidly, enabling a fast delivery of interrupts to the partitions. Furthermore, it improves the performance of hypercalls since they are issued by a software interrupt and lowers the overhead caused by the hypervisor's scheduling tick.

The lack of cache memories and intricate performance optimizations in processors for deeply-embedded systems has several benefits. Processing operations usually execute in the same number of clock cycles and the performance can

Table 1: Execution time of frequently used hypervisor functionality.

	cycles
Scheduling and virtual timer handling	165
Deliver IRQ from thread	212
Return from IRQ to thread	144
Return from IRQ to next IRQ	151
Queue send 4 byte	367
Queue receive 4 byte	303
RPC 4 byte (send + reply)	887

be easily predicted. This is in contrast to an application processor, where non-linear effects have to be taken into account when a para-virtualization approach is used. In such a system, frequent intervention of the hypervisor can lead to increased cache contention and the performance does not scale linearly with the clock frequency due to limited memory latency and bandwidth. Due to these characteristics of deeply-embedded systems, the overhead of using para-virtualization is significantly smaller and more predictable than in other systems.

4.1 Hypervisor

The execution time of important hypervisor functionality is shown in Table 1. The scheduling time determines how short the timeslices of the cyclic scheduling scheme can be made, without resulting in a large overhead. Shorter timeslices allow for better interrupt latency and an improvement in the proportionality of the partitions progress.

The time to enter and return from an interrupt is crucial for systems that handle a large number of interrupts. Furthermore, it is important for the virtualization of operating systems that issue system calls by the use of software interrupts. The total overhead of approximately 350 cycles can have a noticeable effect on the latency of interrupts. Since it takes less time to dispatch an exception than to dispatch a thread, the latency is however lower than the time to deliver an event from hardware to thread level in a partitioning operating system. Whereas, the virtualization approach does not reach the IRQ handling efficiency of a federated system, it is usually more efficient than a traditional partitioning operating system.

A 4-byte RPC creates a total overhead of 887 cycles, including send and reply phase. Using a queue, two send operations and two receive operations would be required to establish a communication between client and server. In our setup, RPC communication is therefore approximately 35% faster than the use of queues. In addition, the use of RPC communication increases the timing predictability and simplifies the implementation of the client.

4.2 Guest OS

We ported the FreeRTOS [1], which is fairly representative for the class of RTOSs used in deeply-embedded applications, to the vCPU interface of our hypervisor. To evaluate the performance of FreeRTOS as guest OS, we measure the number of system calls that are performed in fixed duration in a federated architecture with two processors and compare it to the number that are performed in a virtualized system. To test the partitioning approach in a realistic use case, two instances of FreeRTOS are executed in parallel on the hypervisor. In order to keep the total processing power in both systems constant, the reduction in the number of processors is compensated by an increase of the clock frequency.

Table 2: FreeRTOS systemcall performance in a federated and virtualized architecture with the same processing power. The timeslice length of the hypervisor is 5 ms. Normalized to federated architecture. Higher is better.

	Federated	Virtualized
Number of Processor	2	1
Clock Frequency	14 MHz	28 MHz
Cooperative Thread Switching	100	78.6
Preemptive Thread Switching	100	95.7
Queue send + receive	100	104.7
Queue preemption	100	102.6

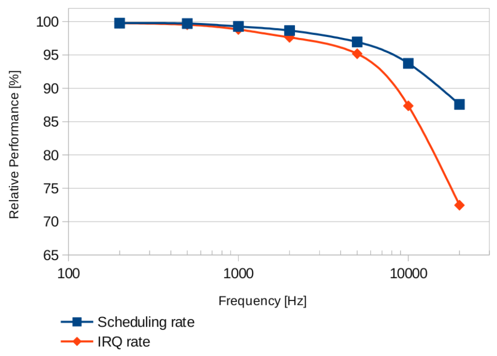

Figure 8: Effect of the hypervisors scheduling rate and the IRQ frequency in the guest OS on performance. Compared to a federated system.

As can be seen in Table 2, the performance numbers vary only marginally and simply reflect small differences in the architecture specific part of the guest OS. The largest difference can be seen for cooperative thread switching. On a Cortex-M3 MCU, thread switching is performed by a special exception and the MCU automatically stores and restores a part of the context. Because the use of exceptions would require a costly interaction with the hypervisor, we use a purely software-based thread switching routine which has a larger overhead. The overhead is clearly visible in the extremely fast cooperative thread switching, but much less pronounced in the slower, preemptive thread switching operations.

Most system calls for communication and synchronization even perform better on the virtualized system. The reason is that the use of critical sections is slightly faster on the virtual processor than in the native system. This is due to the fact that setting a single bit in memory is faster than a modification of the processors interrupt mask [19]. Since the total overhead of the RTOS accounts only for a small fraction of the processor utilization in a realistic system, we expect an almost unaltered performance in a federated and virtualized system.

The performance can however be affected, if high IRQ rates have to be supported. In order to support IRQs with high arrival rates and short latencies in a reservation-based system, it is necessary to frequently make a scheduling decision. To evaluate the performance reduction caused by the hypervisor's scheduling, we reduce the length of the timeslice step-wise from 10ms to 50us and measure the performance of a thread in the guest OS. However, a timeslice length of 50us

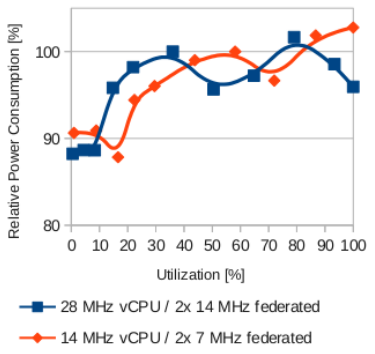

Figure 9: Power consumption of the virtualized system normalized to a federated system with two processors. The processing power of the systems is equal.

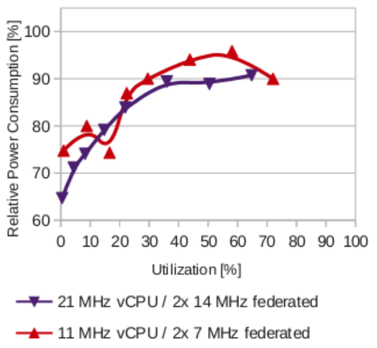

Figure 10: Significant amounts of power are saved when the virtualized system has a lower processing power than the federated system.

is an extremely demanding test case and a length of 500us (corresponding to a scheduling rate of 2kHz) will easily suffice for the bulk of embedded systems. Figure 8 shows that at a scheduling rate of 2kHz the performance loss is merely 1.3%, although our test system is an ultra-low-power MCU which runs at a clock frequency of only 28MHz.

Decreasing the timeslice length below ranges of 1ms can lead to severe performance degradation in application processors that have a much higher performance. It is possible in our system, since there are no indirect costs caused by increased cache contention, the IRQ latency of the processor is low and since the scheduling algorithm is lightweight.

As the scheduling rate increases, it is possible for the partitions to handle IRQs with higher frequencies. Assuming a system has two partitions and a timeslice length of 500us, each partition is scheduled once per millisecond and can therefore handle IRQs with rates of 1kHz. The systems total IRQ rate is 2kHz. Then, an additional overhead is caused by the hypervisor during IRQ delivery and the system performance is decreased to 97.6% of a federated system.

At even higher IRQ frequencies a significant performance loss occurs. If an IRQ with very high frequency has to be handled, it could be executed below the level of the application, in parallel to the level of the hypervisor, thus avoiding the virtualization overhead. The resulting system would not provide the same level of fault containment, leading to an increased effort for safety certification. However, this point illustrates that the virtualization approach is not limited to systems with average or low IRQ rates.

4.3 Power Consumption

We measure and compare the power consumption of a federated system, consisting of two processors, with the corresponding virtualized system. The power consumption of the federated system is measured at clock frequencies of 7MHz and 14MHz. Each processor executes the same task, that is triggered at fixed intervals of 10ms by an external IRQ. When the task has finished processing, the processor enters the idle mode. We modify the load of the task and measure the utilization and power consumption of the processors.

Using our hypervisor, the federated system is integrated on a single MCU with an increased clock frequency. In this setup, the hypervisor uses a scheduling period of 5ms.

The supply voltage of the processor is kept at 3V in both, the federated and integrated architecture. A higher amount of bus communication is necessary in the federated system, that leads to additional runtime overhead and power consumption, which is not considered in this setup.

The comparison is mainly affected by two factors. The overhead of the hypervisor, which causes an increase in execution time and dynamic power consumption. And the reduction to one processor, which leads to a reduction of the static power consumption (mainly leakage power). Since we use a low-power MCU which is optimized for minimal static power consumption, higher power savings can be expected in other systems.

The relative power consumption of the virtualized system compared to the federated system is shown in Figure 9. The behaviour is surprisingly complex. At large utilizations, the total power consumption is high, and neither the hypervisor's overhead nor the amount of static power plays a large role. Instead, the results seem to be dominated by small variations in the processors efficiency. In total, the virtualized system is slightly more power efficient than the federated architecture.

As the utilization of the system decreases, the impact of the static power consumption as well as the hypervisor's overhead increase. The result is a race between the two competing factors. Below utilizations of 20%, the power reduction that can be achieved by saving static processor power exceeds the hypervisors overhead. As a result, power savings around 10% are achieved, making virtualization a good choice for systems with low processor utilizations.

Since execution time slack can be transferred by the timeslice donation policy and since the overhead of handling bus communication is reduced, the processing performance of the virtualized system can often be chosen smaller than in a federated system. We test two virtualized systems whose processing performances are approximately 25% smaller (Figure 10), resulting in power savings of up to 35%.

5. RELATED WORK

The use of virtual processors is most notable in the Xen project [4] [20]. Recently, vCPUs were applied for virtualization in the L4 microkernel [10]. L4 microkernels are known for their high inter-process communication (IPC) performance [11]. The basic concept of RPC used in this work is derived from L4's IPC mechanism, but extended for communication between virtual processors and strong temporal separation.

Several works cover the problem of sharing resources in a reservation-based scheduling scheme, usually by use of the priority-ceiling-protocol [5][15][18]. In our approach, resources are shared by the use of a RPC mechanism that uses a critical section for synchronization. The method proposed by us is comparatively simple, resulting in longer response times to events, but also in very low runtime overhead and predictable timing behaviour. It is therefore well-suited for the use in deeply-embedded systems.

XtratuM [13], is a hypervisor for safety-critical systems. It is following the ARINC 653 [2] standard for avionic architectures, which requires a relatively high overhead for resource sharing. The SPIRIT μKernel [9] provides strong partitioning to legacy RTOSs by use of an Event Delivery Object, which is similar to a vCPU and results in a very efficient virtualization. Unfortunately, SPIRIT does not seem to be continued.

None of the approaches known to us target deeply-embedded systems nor systems with strong power-constraints.

6. CONCLUSION

We present a hypervisor for partitioned deeply-embedded systems, enabling to improve safety and security architectures and reuse certified software in a new context. Since power consumption is frequently a critical constraint in these systems, we focus on solutions with a minimal runtime overhead. A federated system which handles demanding IRQ rates of 2kHz was ported to the hypervisor, resulting in a performance reduction of only 2.4%.

Due to the properties of deeply-embedded systems, such as simple operating systems and cache-less processors, paravirtualization has a smaller performance overhead and better predictability than in more complex systems. Furthermore, compared to federated systems, it lowers the required processing power and bus communication, leading to a reduced power consumption in most systems. In summary, the use of partitioning and virtualization techniques in deeply-embedded devices deserve higher consideration.

We are currently testing our approach under different system architectures and loads. In the future, we intend to verify it using a safety-critical industrial system.

7. REFERENCES

[1] The FreeRTOS Project. www.freertos.org.

[2] ARINC Specificiation 653, Avionics Application Software Standard Interface. Technical report, Airlines Electronic Eng. Comitee, 1997.

[3] M. A. Awan and S. M. Petters. Enhanced race-to-halt: A leakage-aware energy management approach for dynamic priority systems. In *Proceedings of the 2011 23rd Euromicro Conference on Real-Time Systems*, ECRTS '11, pages 92–101, 2011.

[4] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 164–177, 2003.

[5] M. Behnam, I. Shin, T. Nolte, and M. Nolin. SIRAP: A synchronization protocol for hierarchical resource sharing. In *Proceedings of the 7th ACM & IEEE International Conference on Embedded Software (EMSOFT'07)*, pages 279–288, October 2007.

[6] A. Bilgic, F. Bruns, V. Pichot, and M. Gerding. Low-power smart industrial control. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, March 2011.

[7] V. Chandra and R. Aitken. Impact of technology and voltage scaling on the soft error susceptibility in nanoscale CMOS. In *Proceedings of the 2008 IEEE International Symposium on Defect and Fault Tolerance of VLSI Systems*, DFT '08, pages 114–122, 2008.

[8] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the thirteenth ACM symposium on Operating systems principles*, SOSP '91, pages 122–136, 1991.

[9] D. Kim, Y.-H. Lee, and M. Younis. SPIRIT- μkernel for strongly partitioned real-time systems. In *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*, pages 73 –80, 2000.

[10] A. Lackorzynski, A. Warg, and M. Peter. Generic virtualization with virtual processors. In *Twelfth Real-Time Linux Workshop*, October 2010.

[11] J. Liedtke. Improving IPC by kernel design. *SIGOPS Oper. Syst. Rev.*, 27(5):175–188, Dec. 1993.

[12] C. Lin and S. A. Brandt. Improving soft real-time performance through better slack reclaiming. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, RTSS '05, pages 410–421, 2005.

[13] M. Masmano, I. Ripoll, A. Crespo, and J. J. Metge. XtratuM: a Hypervisor for Safety Critical Embedded Systems. In *11th Real-Time Linux Workshop*, 2009.

[14] M. D. Natale. Moving from federated to integrated architectures in automotive: The role of standards, methods and tools. In *Proceedings of the IEEE*, volume 98, pages 603–620. April 2010.

[15] D. d. Niz, L. Abeni, S. Saewong, and R. R. Rajkumar. Resource sharing in reservation-based systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, RTSS '01, pages 171–, 2001.

[16] D. E. Porter, S. Boyd-Wickizer, J. Howell, R. Olinsky, and G. C. Hunt. Rethinking the library OS from the top down. *SIGARCH Comput. Archit. News*, 39:291–304, March 2011.

[17] J. Rushby. Partitioning for safety and security: Requirements, mechanisms, and assurance. Technical report, NASA Langley Research Center, 1999.

[18] S. Saewong, R. R. Rajkumar, J. P. Lehoczky, and M. H. Klein. Analysis of hierarchical fixed-priority scheduling. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, ECRTS '02, pages 173–, 2002.

[19] D. Stodolsky, J. B. Chen, and B. N. Bershad. Fast interrupt priority management in operating system kernels. In *USENIX Symposium on USENIX Microkernels and Other Kernel Architectures Symposium - Volume 4*, moas'93, 1993.

[20] S. Xi, J. Wilson, C. Lu, and C. Gill. RT-Xen: Towards real-time hypervisor scheduling in Xen. In *Proceedings*

of the ninth ACM international conference on Embedded software, EMSOFT '11, pages 39–48, 2011.

[21] D. Zhu, R. Melhem, and D. Mosse. The effects of energy management on reliability in real-time embedded systems. In *Proceedings of the 2004 IEEE/ACM International conference on Computer-aided design*, ICCAD '04, pages 35–40, 2004.