SPIRIT-μKernel for Strongly Partitioned Real-Time Systems

Daeyoung Kim and Yann-Hang Lee

Real Time Systems Research Laboratory
CISE Department, University of Florida
P.O. Box 116120
Gainesville, FL 32611-6120, USA
{dkim, yhlee}@cise.ufl.edu

Mohamed Younis

Honeywell International Inc.,
Advanced Systems Technology Group
9140 Old Annapolis Road
Columbia, MD 21045, USA
mohamed.younis@honeywell.com

Abstract

To achieve reliability, reusability, and cost reduction, a significant trend of building large complex real-time systems is to integrate separated application modules. An essential requirement of integrated real-time systems is to guarantee strong partitioning among applications. In this paper we propose a microkernel, called SPIRIT-μKernel, for strongly partitioned real-time systems. The SPIRIT-μKernel has been designed and implemented based on a two-level hierarchical scheduling methodology such that the real-time constraints of each application can be guaranteed. It provides a minimal set of kernel functions such as address management, interrupt/exception dispatching, inter-application communication, and application scheduling. To demonstrate the feasibility of the SPIRIT-μKernel, we have ported two different application level real-time operating systems (RTOS), WindRiver's VxWorks and Cygnus's eCos, on the top of the microkernel. The SPIRIT-μKernel architecture is practical and appealing due to its low overheads of kernel services and the support for dependable integration of real-time applications.

1. Introduction

Recently integrated real-time systems have been the subject of significant research in both industry and academia. In an integrated real-time system, applications of diverse levels of temporal and mission criticality are supposed to share the same computing resources while maintaining their own functional and temporal behaviors. To protect applications from potential interference, the integrated system must provide spatial separation such that the memory and I/O space of an application are protected from illegal accesses attempted by other applications. In addition, the system must support temporal separation such that the execution time reserved to one application would not be changed due to either execution overrun or events of other applications. We refer to such kind of systems as Strongly Partitioned Real-Time Systems (SP-RTS).

The concept of Integrated Modular Avionics (IMA) [1], which is being braced by the aerospace industry these days, is a good example of SP-RTS. With the strong partitioning support in IMA systems, the same processor can run crucial applications such as control navigation systems and a non-flight-critical function such as cabin's temperature control. Even if the software that controls the cabin temperature is not certified to the highest level and has a probability of not working correctly, the navigation system would not be affected. By integrating applications from federated computer boxes into smaller number of high performance sharable computing resources, they can reduce the interconnection network, weight, power supplies, and physical volumes of computing resources. This also achieves the reliability, maintainability, and dramatic cost reduction. Also a commercial-off-the-shelf (COTS) approach, which is one of the key benefits of SP-RTS, contributes a lot in the development and maintenance of the system considering long-life time of such systems.

In this paper, we propose a SPIRIT (Strongly Partitioned Integrated Real-tIme sysTem)-μKernel that is a key technology in implementing SP-RTS. In designing the SPIRT-μKernel, we followed the design concepts of the second-generation microkernel architecture because it provides good reference model to achieve both flexibility and efficiency.

The goals of the SPIRIT-μKernel are to provide dependable integration of real-time applications, flexibility in migrating operating system personalities from kernel to user applications including transparent support of heterogeneous COTS RTOS on top of the kernel, high performance, and real-time feasibility. To support integration of real-time applications that have different criticality, we have implemented strong partitioning concept using a protected memory (resource) manager and a partition (application) scheduler. We also developed a generic RTOS Port Interface (RPI) for easy porting of heterogeneous COTS real-time operating

systems on top of the kernel in user mode. The satisfactory flexibility and efficiency are achieved by adopting design concepts of the second-generation microkernel architecture. The kernel provides minimum set of functions such as address space management, interrupt/exception dispatcher, inter-partition communication, and partition scheduler, etc. A variety of operating system personalities such as task scheduling policy, exception-handling policy, inter-task communication can be implemented within the partition according to individual requirements of partition RTOS. To demonstrate this concept, we have ported two different application level RTOS, WindRiver's VxWorks 5.3 and Cygnus's eCos 1.2, on top of the SPIRIT-μKernel.

The scheduling policy of the SPIRIT-μKernel is based on our previous work of the two-level hierarchical scheduling algorithm which guarantees the individual timing constraints of all partitions in an integrated system [2,3,4]. At the lower microkernel level, a distance constrained cyclic partition scheduler arbitrates partitions according to an off-line scheduled timetable. On the other hand, at the higher COTS RTOS level, each local task scheduler of a partition schedules own tasks based on a fixed priority driven scheduling. In our previous paper [5] we advocated an approach for supporting temporal partitioning and software reuse in IMA system and introduced a software architecture in which we addressed time management and privilege instruction emulation. But, the previous approach has relative weak points. First, since the time management requires high-precision timer and reference clock, it cannot be applied to systems that do not support both timer and reference clock. Second, emulation of privilege instruction causes frequent kernel-user mode switches and overhead to process the privilege violations. We solve the two problems with low-overhead kernel-tick method and generic RTOS Port Interface (RPI). We also address details of SPIRIT-μKernel in viewpoint of operating system.

The rest of the paper is structured as follows. We discuss SP-RTS model and design concepts of SPIRIT-μKernel in section 2. We describe the kernel architecture in section 3. In section 4, we present generic RTOS Port Interface. In section 5, we explain the prototype implementation and performance evaluation. A short conclusion is then followed in section 6.

2. SP-RTS Model and Design Concepts

Strongly partitioned real-time system is a system architecture which supports the integration of multiple, different levels of temporal and mission criticality for real-time applications in sharable computing environment. These integrated applications must be guaranteed to meet their own timing constraints while sharing resources like processors, communication bandwidth with other applications. To guarantee timing constraints and dependability of each application, it is required to realize the concepts of strong partitioning, spatial and temporal partitioning.

2.1. Strongly Partitioned Real-Time System Model

The SP-RTS is composed of multiple communicating partitions in which there are also multiple interacting tasks as shown in Figure 1. The SP-RTS uses the two-level hierarchical scheduling policy in which partitions are scheduled by the SPIRIT-μKernel's cyclic partition scheduler and the tasks of a partition are scheduled by the fixed priority driven local task scheduler of each partition.

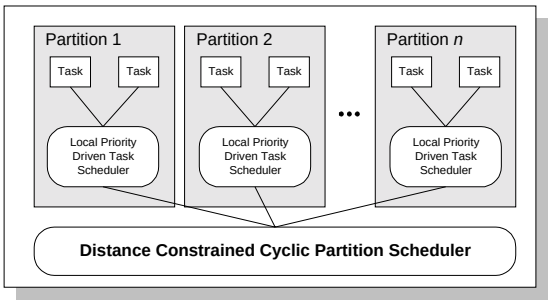

Figure 1. SP-RTS Model

In order to meet the deadlines of periodic tasks within each partition, the processing capacity dedicated to a partition is then dependent on how frequently it is served in the cyclic schedule. In [2,3], we have investigated the constraints on the allocated capacity (α) and the invocation period (h) and been able to construct a monotonic function between α and h for any set of periodic tasks under either rate- or deadline-monotonic scheduling algorithms. Then, similar to pinwheel scheduling [10], we can compose a distance-constrained cyclic schedule for all partition in the SP-RTS system. Following this approach, for a system of n ordered partitions $P_{1,\ldots,n}$, each partition server, P_i, will be given a processor capacity of α_i in the period of h_i to process its periodic tasks, where $\sum_{i=1}^{n} \alpha_i \leq 1$ and h_i | (divides) h_j for $i < j$. By the characteristics of distance constrained cyclic scheduling, the independence between lower level partition scheduler and higher level local task scheduler is achieved.

2.2. Design Concepts of SPIRIT-μKernel

As a key implementation vehicle of the strongly partitioned real-time systems, the SPIRIT-μKernel has the following design concepts at its core:

Ensuring temporal partitioning - Enforcing temporal separation should be implemented efficiently by avoiding excessive overhead and feasibly by meeting application's timing constraints. The key to achieve temporal partitioning is the scheduling strategy. This is one of the

most difficult jobs to build a feasible schedule for the SP-RTS. In our previous work [2,3], we have devised feasible two-level hierarchical scheduling algorithm that solves a pair of fixed priority and cyclic scheduling. In the first prototype of the kernel, we adopt this pair of scheduling policy.

Ensuring spatial partitioning - Resources allocated to a partition must be protected from unauthorized access by other partitions. Also the kernel's system resources must be protected from the partitions. In addition to protection, efficiency and deterministic applications execution are important features that should not be sacrificed for spatial partitioning.

Supporting applications based on heterogeneous COTS RTOS on top of the kernel - Our emphasis is also in the flexibility of accommodating COTS real-time operating systems on top of the kernel. For example, in Integrated Modular Avionics, there is a need to integrate multiple applications that are originally developed in different real-time operating systems. If the kernel and scheduling algorithms are certifiable, it can reduce tremendous amount of integration efforts and costs.

Restricting COTS RTOS to user mode - In the SP-RTS all codes including COTS RTOS kernel of the partition must be run in user mode. This is a challenge because we need to develop a secure interaction method between a kernel and partitions and to modify COTS real-time operating systems in order to enforce such a policy.

Capturing second-generation microkernel architectural concepts - Since the SP-RTS supports different kinds of application environments in a shared computing platform, it is better to follow microkernel architecture than monolithic kernel. The second-generation microkernel architecture provides better flexibility and high performance.

3. SPIRIT-µKernel Architecture

The SPIRIT-µKernel provides strongly partitioned operating environment to the partitions that can accommodate application specific operating system policies and specialties. A partition can have a flexibility of choosing its own operating system personalities such as task scheduling policy, interrupt and exception handling policy, and inter-task communication, etc. The SPIRIT-µKernel provides only minimal necessary functions as shown in Figure 2. Like second-generation micro-kernel architecture, current implementation of the SPIRIT-µKernel takes advantages of hardware support from the PowerPC processor architecture. We believe that the kernel can be ported to different hardware platform easily because the kernel requires only a MMU and timer interrupt support. In this section, we discuss only the fundamental features of the kernel. The RTOS Port Interface will be discussed in the following sections.

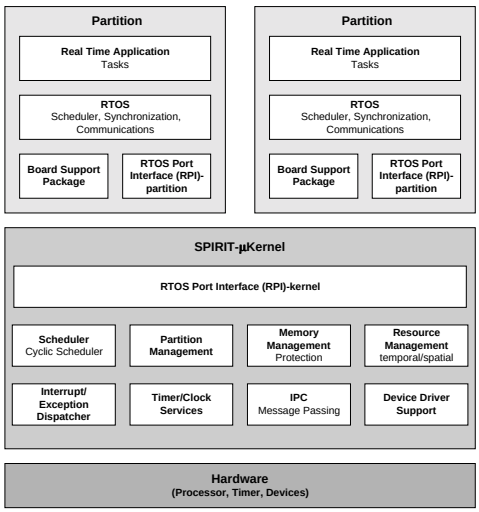

Figure 2. The Architecture of SPIRIT-µKernel

3.1. Memory Management

To guarantee strong spatial partitioning, we use fixed two-level hierarchy of address spaces, kernel and partition address spaces. The kernel and partitions have their own protected address space in which their codes, data, and stacks are located. Protection of address space is implemented by a hardware protection mechanism like the memory management unit of a processor. We believe that an optimization of address space management to the underlying processor architecture is the best solution to achieve both efficiency and deterministic access. A kernel address space is protected from illegal accesses of partitions and a partition address space is also protected from potential interference of other partitions. Since the kernel is dependable and needed to be certified in safety critical systems, we can allow the kernel to access the address spaces of all partitions.

In the SPIRIT-µKernel environment, address space allocation and access policy are determined at system integration time and saved in secure kernel area. Like most real-time operating systems, we excluded the use of virtual memory concept in which each partition is allocated independent virtual address space. While configuring an integrated system, we divide total physical memory space into non-overlapped regions that satisfy the memory requirement of each partition. Instead of using full features of the memory management unit, we use an address translation from virtual (logical) address to physical address for protection purpose only. If we allocate a separate page table per partition for supporting full virtual memory system, we would face poor deterministic behavior and performance, and increased page table size. In the SPIRIT-µKernel, since we use the same address for both virtual address and physical address,

the page table size is only dependent on the size of physical memory.

To achieve high performance as well as protection, we optimized the memory management scheme of the SPIRIT-μKernel according to PowerPC architecture. Both MMU BAT (Block Address Translation) and Segment/Page functions of the PowerPC are used to implement spatial partitioning. The BAT and Segment/Page table mechanisms are used to protect kernel's address space and partition's address space respectively. Combined use of the two memory management techniques of the PowerPC enables us of achieving very low overhead in kernel-partition switching and task switching within the partition address space.

3.2. Partitions Management

An application partition is the basic scheduling entity of the SPIRIT-μKernel. A partition represents a well defined and protected, both temporally and spatially, entity in which different kinds of real-time operating systems and their applications can be implemented. Different operating system personalities such as local task scheduler, communication facilities, synchronization method, etc., can be implemented within the boundary of a partition. Due to the two-level static scheduling policy used in our SP-RTS, temporal and spatial resources are allocated to the partitions at system integration time. Therefore, all attributes of the partitions including time slices, memory allocations, etc., are stored in partition's configuration area in the kernel space. The examples of this configuration information include address range, cyclic schedule table, value of local time tick slice, addresses for event server, event delivery object, partition entry, and interrupt flag.

Basically we do not allow the shared libraries among the partitions because our goal is to protect against partition failure, even caused by the COTS kernel itself. So all partition code including partition's local kernel are executed in user mode. If we allow partitions to share COTS real-time operating system kernel code and its libraries, a corruption or bug of the shared code affects all the partitions that share the code. However, we allow shared libraries among the non-critical partitions to save the memory space.

As described earlier, we use distance constrained cyclic partition scheduler to dispatch partitions. There are two possible methods in implementing cyclic scheduler. One is to use a SPIRIT-μKernel time tick implemented by periodic time tick interrupt with a fixed rate. In this case, the granularity of the execution time allocation to the partitions and local time tick slice of a partition are limited by the resolution of the kernel tick. Although it may decrease the schedulable utilization bound of the integrated system, it enables building a simpler and more predictable cyclic scheduler. The other approach is to use a reference clock and a timer of fine resolution. It has advantages of accurate time allocation and avoiding spurious kernel tick overhead, however it increases the complexity of managing time both in the kernel and partitions. In the prototype implementation, we use the first approach as used in MIT's Exokernel.

Since the SPIRIT-μKernel does not restrict the scheduling policies of the kernel scheduler and local task scheduler of a partition, it is easy to use other hierarchical scheduling policies in SPIRIT-μKernel environment.

3.3. Timers/Clock Services

Timers/clock services provided by COTS RTOS generally rely on the time tick interrupt of the kernel, which has relatively low resolution such as 10ms, or 16ms. The sleep system call is a typical example of this service category. The other way of providing high-resolution timers/clock service can be implemented by special timer device and its device driver.

A partition maintains local time-tick that is synchronized to the global reference clock, the value of kernel tick. The local time tick interval of a partition must be multiples of kernel tick interval. Since we use cyclic scheduling approach, it is possible that user timer service event be occurred in the deactivation time of a partition. The solution is that every time a partition is resumed, it goes to local event server first. The event server checks the events that have occurred during the deactivation time of the partition and delivers events to the local kernel if needed. The possible delay of an event delivery can be solved by our previous scheduling works [2,3]. The event server will be discussed later in the paper

3.4. Inter-Partition Communication

In complex and evolving real-time systems like SP-RTS, it is very important to provide scalable and robust communication facilities for communicating partitions. The legacy inter-process communication method used in traditional microkernel architecture is not sufficient enough to support all requirements of SP-RTS which needs special features like fault tolerance, one (many)-to-many partitions communications, supporting system evolution, and strong partitioning.

Therefore, we have selected a Publish-Subscribe model as basic inter-partition communication architecture of SPIRIT-μKernel. The proposed Publish-Subscribe communication model was designed to achieve following goals:

Supporting a variety of communication model – Due to its complexity of SP-RTS, it needs a variety of communication models such as point-to-point, multicast, broadcast to serve various requirements of integrated partitions.

Fault tolerance – In SP-RTS, it is not uncommon to replicate applications for fault tolerance. With enhanced

Publish-Subscribe model we can implement fault tolerant communication model like master/shadow model.

Evolvability/upgradability – Considering relatively long -lifetime of complex real-time systems such as avionics, it is easily anticipated that the system will face modifications for system evolution and upgrade. A basic Publish-Subscribe model is a good candidate for this purpose.

Strong partitioning – Basically, all communication requirements are scheduled statically at system integration time. So the correctness and compliance of messages exchanged among partitions are to be checked at the kernel to protect other partitions from being interfered by non-compliant messages.

More detailed information about inter-partition communication is beyond the scope of the paper.

3.5. Device Driver Model

The SPIRIT-µKernel provides two device driver models for exclusive and shared devices. The exclusive device driver model can be used for the devices that are safety-critical or exclusively used by a partition. To allow a partition to access exclusive devices, the device I/O address must be exclusively mapped to the address space of the corresponding partition. In safety critical systems like avionics system, it is preferred to use polling method than interrupt method because polling method is more predictable and reliable than interrupt method. The shared device driver model is used for devices that are potentially shared by multiple partitions using multiplexing /demultiplexing methods. More detailed information about device driver model is beyond the scope of the paper.

4. Generic RTOS Port Interface (RPI)

The SPIRIT-µKernel provides generic RTOS Port Interface (RPI) that can be used to port different kinds of COTS real-time operating systems on top of the kernel. With the help of RPI, the SPIRIT-µKernel itself can be built independently without considering COTS RTOS. In this section, we describe the selected features of RPI such as Event Delivery Object (EDO), Event Server, user-mode Interrupt Enable/Disable emulation, and Kernel Context Switch Request (KCSR) primitive. To give an overview of RPI mechanism, we depict an example of partition switching using RPI in Figure 3. In Figure 3, when a partition is to be preempted, the interrupt/exception dispatcher saves the context of current partition and loads the context of next partition (1). The dispatcher prepares EDO including new loaded context and delivers it to the event server of the next partition (2). The event server does housekeeping jobs for the next partition and also invokes the scheduler of the partition if needed (3). The updated partition context, to which CPU control goes, is delivered to the SPIRIT-µKernel via Kernel Context Switch Request primitive (4). Finally, the kernel gives CPU control to proper location informed by KCSR primitive (5).

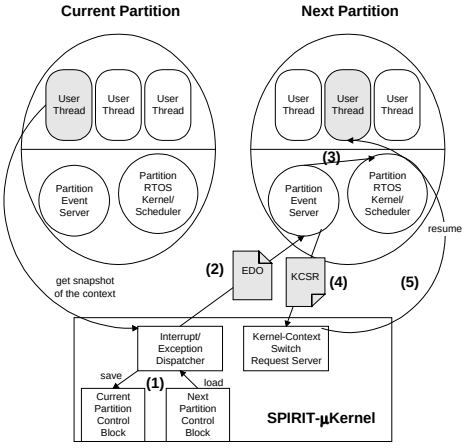

Figure 3. EDO, Event Server, and KCSR Interaction

4.1. Event Delivery Object (EDO)

A partition requires to be informed of hardware events such as time tick expiration, processor exceptions, and interrupts. Since these events are caught in *SPIRIT-µKernel* in supervisor mode, secured and efficient delivery of an event from the kernel to a partition is necessary to be processed in local event handling routine of a partition. In SP-RTS, it is not allowed for the kernel to directly execute RTOS-specific local event handling routines for two reasons. First, if an event handler in RTOS's address space is called in the context of the kernel (supervisor mode), it may violate the strong partitioning requirements of the system due to the possible corruption or overrun of the event handler. Second, it increases the complexity of kernel's interrupt/exception handling codes and data structures because the kernel must have detailed knowledge of the RTOS design, which is vendor's proprietary in COTS RTOS. So we have devised generic event delivery method which can be used to solve the problem. The Event Delivery Object (EDO), along with event server and kernel context switching request primitive, is one of the key components building generic RTOS port interface.

The EDO is physically located in the address space of a partition, so it can be accessed from both the kernel and its owner partition. We show the abstract structure of EDO that is used in our PowerPC prototype in Figure 4.

Based on the EDO mechanism, we have developed the generic interrupt and exception handling routine of the SPIRIT-µKernel shown in Figure 5. We describe the usage of EDO with an example of task switching caused by partition's local time tick expiration.

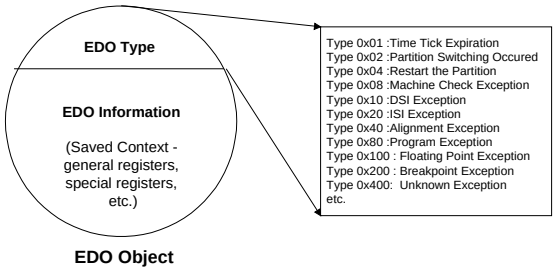

Figure 4. The Structure of Event Delivery Object

Response to local time tick expiration, the local task scheduler may switch tasks based on its local scheduling policy and the status of tasks. In Figure 5, upon the arrival of time tick expiration interrupt, the kernel saves the context in the EDO information area of corresponding partition and sets EDO type as time tick expiration (7,8). After preparing an EDO, the kernel sets interrupt return address as the entry address of the partition's Event Server (9).

After return from the interrupt, now in user mode, the local event server checks the EDO and dispatchs the proper local event handling routine. In VxWorks case, receiving a time tick expiration, the local time tick interrupt handler checks the status of the VxWorks kernel whether task switching is needed or not. When a task switching is needed, it saves the context in EDO information area in the task control block of the preempted task. Then the event server issues a kernel context-switch request to the kernel providing the context of newly dispatched task. Since all local event-handling routines are executed in user mode and in the local partition's address space, we can protect the kernel and other partitions from the faults of a partition. The overall procedure of the event server can be found in Figure 6.

4.2. Event Server

The event server of a partition is an essential part of the SPIRIT-μKernel environment. It has a role of a mediator between the kernel and partitions by executing partition's local interrupt and exception handlers and performing housekeeping jobs such as delivering local kernel events which occurred during the deactivation time of a partition. Figure 6 shows the generic event server procedure of a partition. The implementation of an event server depends on corresponding RTOS. We explain the handling of partition switching event as an example.

In Figure 6, when an event server is user-mode-called from the SPIRIT-μKernel, it checks the type of EDO and process the EDO properly. If the event is a partition switching, it executes housekeeping jobs of a partition (3). Since there may be asynchronous local kernel events that were arrived in the deactivation time of the newly dispatched partition, they must be delivered to the local RTOS kernel before dispatching the task of the partition. If rescheduling is needed in local RTOS kernel and current resume pointer is within application task not within a local kernel, the task context which is in EDO will be saved in task control block and new context will be transferred to the kernel using KCSR primitive (4,5).

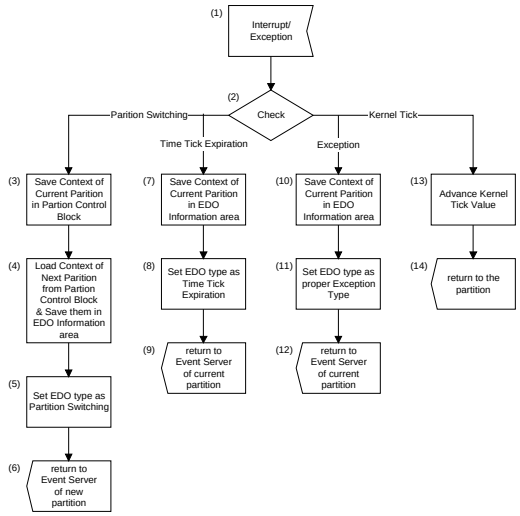

Figure 5. Interrupt/Exception Handling Procedure

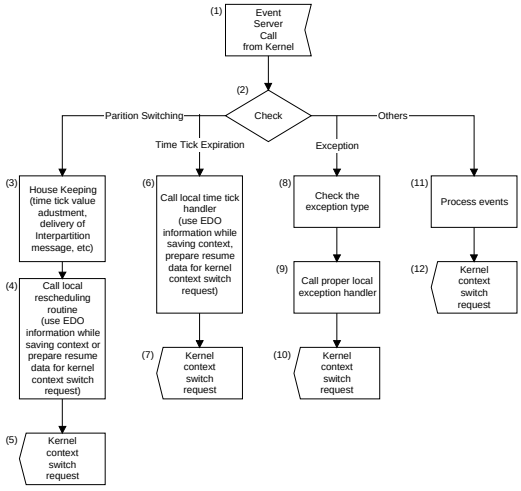

Figure 6. Event Server Procedure

4.3. Kernel Context Switch Request Primitive

In most processor architectures, the context of a task is composed of both user-mode and supervisor-mode accessible registers. When a partition's local task scheduler tries to restore the context of next dispatched task in SPIRIT-μKernel system, privilege violation exception may occur because context loader runs in user mode. To solve this problem, we have devised kernel context switch request primitive, which delegates the kernel to perform a context-loading job. The kernel is

provided restoring context information with which the kernel loads and resumes a new task.

4.4. Interrupt Enable/Disable Emulation

Most COTS real-time operating systems use interrupt-enable and disable functions to guard critical sections in implementing system calls and the core of their kernels. But, in the SPIRIT-μKernel environment, we cannot allow a partition to execute interrupt control functions because these functions must be run in supervisor mode. To solve this problem, it is required to re-write original interrupt control functions while guaranteeing its original intention of protecting critical sections from interrupts.

We use atomic set and reset instructions to implement replacements of original interrupt enable and disable functions. In PowerPC architecture, atomic set and reset logical functions are implemented by a reservation-based instruction set. Disabling interrupts is emulated by atomic set of interrupt flag, which is located in the partition address space and shared by the kernel and owner partition. While the interrupt flag is set, kernel delays the delivery of an interrupt until the interrupt is enabled. The interrupt enable function is implemented by atomic reset instruction that clears the flag.

5. Performance Evaluation

In order to prove and demonstrate the feasibility of the proposed SPIRIT-μKernel, we have implemented a prototype on DY4-SVME171 commercial-off-the-self embedded CPU board. The board houses an 80MHz Motorola PowerPC 603e and 32MB of main memory. On top of the hardware platform, we have implemented the SPIRIT-μKernel that cyclically schedules heterogeneous COTS RTOS partitions, Windriver's VxWorks and Cygnus's eCos. The size of the SPIRIT-μKernel code is 36 Kbytes. In evaluation prototype we integrate and run four different partitions, two VxWorks-based and two eCos-based applications. We measure the execution time using PowerPC time base register, which has 120 ns resolution.

5.1. Kernel Tick Overhead

Since the kernel and partition's local RTOS are synchronized to and scheduled based on the kernel tick, the resolution of the kernel tick is important factor that affects the system performance and schedulability. To evaluate the overhead of the kernel tick, we obtained two basic components, *kt_overhead1* and *kt_overhead2*. The *kt_overhead1* is measured when a kernel tick is used for neither partition switching nor local partition time tick expiration. The *kt_overhead2* is the time used to deliver the EDO type of Time_Tick_Expiration to the event server of the local partition. We show the result in Table 1.

Table 1. Measured Kernel Overheads

kt_overhead1			*kt_overhead2*		
Avg	*Min*	*Max*	*Avg*	*Min*	*Max*
0.84 μs	*0.75 μs*	*3.88 μs*	*9.38 μs*	*9.00 μs*	*18.38 μs*

To help find the feasible kernel tick resolution, we calculate *r_kt_overhead* and *p_kt_overhead* that are the percentages of the redundant kernel tick overhead and total kernel tick overhead for partition's local task scheduling respectively.

$$r\_kt\_overhead = \frac{kt\_overhead1}{kt\_period}$$

$$p\_kt\_overhead = \frac{kt\_overhead1 * (\frac{pt\_period}{kt\_period} - 1) + kt\_overhead2}{pt\_period}$$

In Figure 7, we depict average and worst case kernel tick overheads that are obtained by varying kernel tick period, 100us, 500us, 1ms, 2ms, 5ms, with fixed partition local tick period of 10ms.

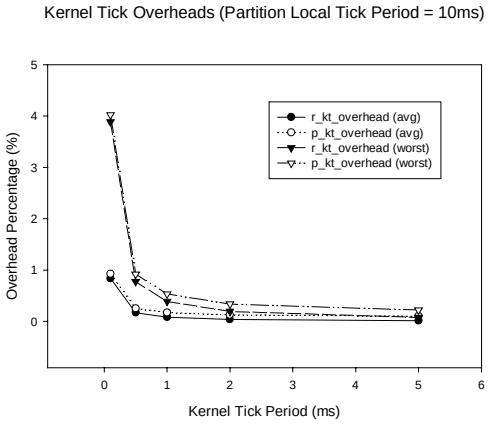

Figure 7. Kernel Tick Overheads

As illustrated in Figure 7, while the kernel tick period is bigger than 500μs, each of all four overheads is less than 1% of the total CPU capacity. These practically acceptable overheads were possible due to the efficient design of the kernel tick interrupt handler and low latency interrupt handling support from PowerPC processor.

5.2. Partition Switch Overhead

Based on the off-line scheduled timetable, the kernel scheduler dispatches partitions. When a partition-switching event occurs, the kernel scheduler saves the context of the current partition and reloads saved context of the next partition. Then CPU control goes to the event server of newly dispatched partition. Since the kernel has a generic partition-switching handler regardless of the

types of RTOS that are saved or restored, we expect deterministic partition switch overheads. The Table 2 shows the partition switch overheads.

Table 2. Partition Switch Overhead

Avg	*Min*	*Max*
10.50 μs	***10.00 μs***	***21.50 μs***

5.3. Kernel-User Switch Overhead

As in conventional operating system, kernel-user mode switch overhead is an important factor for evaluating the performance of the system. To reduce the overhead, we use the BAT and segment/page table schemes for the kernel and a partition respectively. Since both schemes are executed concurrently during the address translation and a result is chosen according to the current processor mode, the only overhead for kernel-user switch is the time needed to set or reset the processor mode bit in the machine state register. So we can claim that the pure overhead due to kernel-user switch while distinguishing the processor modes, supervisor and user mode, is ignorable. Instead of measuring the pure kernel-user switch overhead, we measured kernel-context switch request primitive, which is essential way of CPU control transfer, which requires supervisor-mode privilege, in user mode. The execution time of the primitive is measured and listed in Table 3.

Table 3. KCSR Primitive Performance

Avg	*Min*	*Max*
1.34 μs	***1.25 μs***	***7.50 μs***

5.4. TLB Miss Handling Overheads

In many real-time applications, virtual memory is not supported and supervisor and user modes of execution are not distinguished. However, it is an essential principle to distinguish supervisor and user mode to guarantee spatial partitioning concept. The SPIRIT-μKernel uses PowerPC MMU's segment/page scheme to protect the address space of partitions. Since this scheme relies on virtual to physical address translation, the performance of the TLB (Translation Look-aside Buffer) scheme of the PowerPC is very important. Considering the fairly uniform distribution of memory access in SP-RTS environment due to partitioning, we should pay much attention to the performance of TLB. In the prototype, it requires 8K page table entries for 32Mbytes physical memory. However, PowerPC provides only 64 page table entries for ITLB and DTLB respectively. We measured the TLB miss handling overhead and showed in Table 4.

Table 4. TLB Miss handling Overheads

	Avg	*Min*	*Max*
Instruction TLB	***1.41 μs***	***0.88 μs***	***3.50 μs***
Data TLB Load	***2.43 μs***	***1.63 μs***	***4.25 μs***
Data TLB Store	***2.22 μs***	***1.13 μs***	***4.88 μs***

6. Conclusion

The SPIRIT-μKernel is designed to provide a software platform for Strongly Partitioned Real-Time Systems that have been significantly studied in both academia and industries. Using the kernel, we can achieve better reliability, reusability, COTS benefits, and cost reduction in building complex real-time systems.

For further study, we are planning to enhance the kernel in three directions. First, we will build our own local kernel for a partition instead of using COTS RTOS. Second, we will extend current uni-processor architecture to multiprocessor and distributed real-time computing environment. Third, we will develop other two-layer scheduling strategies instead of cyclic/priority driven scheduler pair while guaranteeing strong partitioning requirements.

References

[1] "Design Guide for Integrated Modular Avionics," ARNIC Report 651, Aeronautical Radio Inc., Annapolis, MD, 1991.

[2] Y. H. Lee, D. Kim, M. Younis, and J. Zhou, "Partition Scheduling in APEX Runtime Environment for Embedded Avionics Software," *Proc. of IEEE Real-Time Computing Systems and Applications*, pp. 103-109, Oct. 1998.

[3] Y. H. Lee, D. Kim, M. Younis, J. Zhou, and J. McElroy, "Resource Scheduling in Dependable Integrated Modular Avionics," *Proc. of IEEE International Conference on Dependable Systems and Networks)*, Jun. 2000.

[4] Y.H. Lee, D. Kim, M. Younis, J. Zhou, "Scheduling Tool and Algorithms for Integrated Modular Avionics Systems," *Proc. of IEEE/AIAA Digital Avionics Systems Conference*, Oct. 2000.

[5] M. Younis, M. Aboutabl, and D. Kim, "An Approach for Supporting Software Partitioning and Reuse in Integrated Modular Avionics," *Proc. of IEEE Real-time Technology and Applications Symposium*, May 2000.

[6] D. Engler, M. Kaashoek, and J. O'Toole Jr., "Exokernel: An Operating System Architecture for Application-Level Resource Management," *Proc. of ACM SIGOPS*, pp. 251-266, Dec. 1995.

[7] J. Liedtke, "On m-Kernel Construction," *Proc. of ACM SIGOPS*, pp. 237-250, Dec. 1995.

[8] J. Liedtke, "Toward Real Microkernels," *Communications of the ACM, Vo. 39, No. 9*, Sep. 1996.

[9] H. Hartig, M. Hohmuth, J. Liedtke, S. Schonberg, J Wolter, "The Performance of μ-Kernel-Based Systems," *Proc. of ACM SIGOPS*, France, Oct. 1997.

[10] M. Y. Chan and F. Y. L. Chin, "General schedulers for the pinwheel problem based on double-integer reduction," *IEEE Trans. on Computers*, vol. 41, pp. 755-768, June 1992.