On the Design and Implementation of Uniprocessor Real-Time Resource Access Protocols*

Lucas Pires Camargo
Software/Hardware Integration Lab
Federal University of Santa Catarina
Joinville, Santa Catarina, Brasil
Email: camargo@lisha.ufsc.br

Giovani Gracioli
Software/Hardware Integration Labb
Federal University of Santa Catarina
Joinville, Santa Catarina, Brasil
Email: giovani@lisha.ufsc.br

***Abstract*—Real-time operating systems (RTOS) should support resource access protocols to bound the maximum delay incurred by priority inversions. The implementation of such protocols must be lightweight, because its performance affects the system schedulability. In this paper, we present an object-oriented design of real-time resource access protocols for uniprocessors aiming at reducing the run-time overhead and increasing code re-usability. We implement the proposed design in an RTOS and measure the memory footprint and run-time overhead of the implementation. By applying the obtained overhead into the schedulability of several generated task sets and four protocols, our results indicate that proper implementation of resource access protocols has a small impact on the schedulability of real-time tasks.**

***Keywords*—Real-time resource access protocols, real-time operating systems, priority ceiling protocol, priority inheritance protocol, stack resource policy**

I. Introduction

Any concurrent operating system (OS) must offer some kind of protocol to control the access to shared resources (*i.e.,* a piece of code, such as data structures, I/O devices, buffers, and so on) by competing tasks, thus ensuring *mutual exclusion* in their respective critical sections [1, 2].

Typically, mutual exclusion is guaranteed by the use of binary semaphores, such as mutexes and suspension-based locks [3]. Therefore, a task willing to enter a critical section must wait until another task, which holds the resource, exits the critical section. This is accomplished by calling the semaphore operations *p* (or wait) and *v* (or signal) before entering and after leaving each critical section, respectively.

Real-time embedded applications also use semaphores to synchronize access to shared resources. However, specific resource access protocols are required to avoid unbounded priority inversions [1]–[3]. For instance, consider a higher-priority task τ_a and a lower-priority task τ_b, sharing a resource R_1. It might happen that τ_b is holding the resource R_1, but is preempted by τ_a. Then, τ_a executes until it tries to enter the critical section. However, τ_a cannot continue, because the resource is already in use by τ_b. Thus, a higher-priority task (τ_a) is blocked by a lower-priority task (τ_b) which can cause unpredictable delays if not properly handled.

In fact, priority inversion has caused many problems in real applications. For example, the mars pathfinder spacecraft reseted several times after landing Mars on July 4^{th}, 1997, resulting in significant delays in capturing scientific data [4, 5]. This resetting was caused by the overran of a lower-priority task due to priority inversion, which triggered a watchdog timer. Fortunately, the developers were able to patch the spacecraft remotely to solve the problem.

Several real-time resource protocols have been proposed to bound priority inversion and take the blocking time into consideration when performing schedulability analyses. These protocols can be divided in those for static schedulers (when task priorities do not change) and for dynamic schedulers (when task priorities may change during execution)[1]. Ceiling- and priority inheritance-based protocols [6] are typically used in static scheduling, while the stack resource protocol (SRP) [7] is typically used in dynamic scheduling.

In this work, we present a design and implementation of resource access protocols for uniprocessors, considering both static and dynamic scheduling approaches. More specifically, we present a design for the Priority Inheritance Protocol (PIP) [6], Priority Ceiling Protocol (PCP) [6], Immediate Priority Ceiling Protocol (IPCP), and SRP [7]. Our design uses object-oriented techniques to maximize the software reuse and minimize the run-time overhead. We have implemented the proposed software design in a real-time operating system (RTOS) and evaluated the memory footprint and run-time overhead in a microcontrolled platform. Our results indicate that the proposed design allows software reuse and low overhead, providing schedulability ratios close to the theoretical ones. In summary, we make the following contributions in this paper:

- We propose an object-oriented design for uniprocessor real-time resource access protocols. The focus is on software reuse and low overhead;
- We implement the proposed design in an RTOS and evaluate the memory footprint of this implementation as well as its run-time overhead on a microcontrolled platform. The maximum obtained overhead is less than 20 µs, considering both *p* and *v* operations;
- A comparison in terms of task set schedulability ratio among the protocols, considering their run-time over-

*Work funded by CNPq/PIBIC.

[1]Note that in this work we only consider resource access protocols for Uniprocessors.

heads. Our results indicate that due to the low overhead, the schedulability ratio remains close to theoretical bounds, proving the efficiency of the proposed design.

The remainder of this paper is organized as follows. Section II presents the system model and reviews the resource sharing protocols used in this work. Section III shows the proposed design and implementation of the protocols. Section IV evaluates the proposed design in an real-time operating system and real hardware platform. Section V presents the related work. Finally, Section VI concludes the paper.

II. System Model and Background

In this work we consider the periodic task model, in which a task set τ is composed of *n* implicit-deadline tasks, $\tau = \{\tau_1, ... ,\tau_n\}$, running on a single-processor. Each task τ_i, where $i \leq n$ and $i \geq 1$, has a period p_i and a worst-case execution time (WCET) e_i. A task τ_i releases a job at every p_i time units. r_i^j denotes the release time of the j^{th} job of τ_i, named τ_i^j. The relative deadline of the task τ_i is equal to its period: $d_i = p_i$. The relation e_i/p_i defines the utilization of a task τ_i, called u_i. The sum of all tasks' utilizations defines the total system utilization ($\sum_{i=1}^{n} u_i$). We assume that tasks suspend only to wait for a lock. We also assume that the task set is scheduled either by the Rate-Monotonic (RM) scheduling, when fixed-priority (FP) policy is used, or by the Earliest Deadline First (EDF), when dynamic-priority policy is used. At run-time, resource access protocols may temporarily raised the tasks priorities.

Tasks share n_r serially-reusable resources, $R_1,\ldots,R_{n_r}$. $N_{i,q}$ denotes the maximum number of times that a job of τ_i accesses R_q. $L_{i,q}$ denotes the maximum critical section length required by τ_i when it uses R_q. The priority ceiling C_q is equal to the priority of the highest-priority task that uses R_q.

A. Resource Access Protocols

This sections presents an overview of the real-time synchronization protocols implemented in this work. The protocols are the Priority Inheritance Protocol (PIP), the Priority Ceiling Protocol (PCP), the Immediate Priority Ceiling Protocol (IPCP), and the Stack Resource Policy (SRP). For a complete overview of such protocols, please refer to [1, 2].

PIP is a classic mechanism for sharing resources in a single-processor with FP scheduling. It aims at avoiding priority inversion by elevating the priority of the tasks that hold a resource, when there are higher priority tasks waiting on the same resource. The priority of the running task is always the maximum priority of the tasks blocked on the resource, if it is higher that the original priority.

PCP is another classic algorithm for controlling priority inversion and bounding blocking time for a task set with shared resources. It differs from PIP in the sense that every resource has a priority ceiling C_j, defined as the maximum priority among all tasks that access the resource. Whenever another task blocks on a locked resource, the owner's priority is temporarily raised to the resource ceiling for the remainder of its critical section. This reduces context-switching overhead and simplifies the implementation in comparison to PIP.

IPCP is a variant of PCP, aiming for performance and ease of implementation. The major difference is that the task owning the resource has its priority raised to the ceiling immediately when it first acquires the resource, and not when another task tries to lock the resource. The main effect of this change is a further reduction of context switching overhead.

SRP provides resource access control for dynamic scheduling policies, such as the EDF scheduler. Additionally to a priority, SRP assigns a preemption level π_i to each task. π_i is static, initialized at the task τ_i creation time, and remains the same for all of its job. The main property is that a task τ_a can only preempt another task τ_b if its preemption level π_a is greater than π_b. Under EDF scheduling, this condition is satisfied when preemption levels are ordered inversely in respect to the tasks' relative deadlines, as in $\tau_a > \tau_b \iff D_a < D_b$.

During execution, every resource is assigned a resource ceiling C_{R_i}. This ceiling is equal to the maximum preemption level of the tasks that would be blocked on the resource, when issuing their maximum request. Therefore, the resource ceiling is a dynamic value, and is a function of available units of the semaphore: $C_{R_i} = max(\pi_i | \mu_i(R_k) > n_k$.

SRP also defines a system ceiling Π_s, which is the maximum current system ceiling between all tasks. This is a global, dynamic parameter that can change at any resource access or release. The main idea of SRP is that if a task is going to be blocked on a resource it cannot access, it will be not even be allowed to preempt execution of other tasks. Also, a task is not allowed to begin execution unless any task that could preempt it would not block waiting on a resource. This is achieved by the SRP Preemption Test: a task is not permitted to be executed unless its priority is highest amongst all ready tasks, and its preemption level is higher than the system ceiling.

III. Design and Implementation of Resource Access Protocols

Figure 1 presents an overview of the proposed software design for the resource access protocols through a UML class diagram. To achieve this final design, we used the Application-Driven Embedded System Design (ADESD) [8] methodology to capture common characteristics related to the PIP, PCP, IPCP, and SRP protocols. In the resulting design, there are in total eight classes: `Synchronizer_Common`, `Semaphore`, `Semaphore_RT`, `Semaphore_SRP`, `Semaphore_Ceiling`, `Semaphore_PCP`, `Semaphore_IPCP`, and `Semaphore_PIP`. Below, we provide a detailed description of each class.

The base class `Synchronizer_Common` offers support for operations common to all synchronization primitives, such as mutexes, semaphores, and condition variables. These operations include, for instance, atomic increment and decrement (*finc* and *fdec*), test and set lock (*tsl*), and interrupt enabling/disabling (*begin_atomic* and *end_atomic*). The class also implements an interface for common thread operations, such as sleep, wakeup, and wakeup all. These thread operations basically put a thread to sleep into the synchronization queue (the `_queue` attribute), wakeup a thread that was sleeping in the queue, and wakeup all threads that were sleeping in the queue. All operations

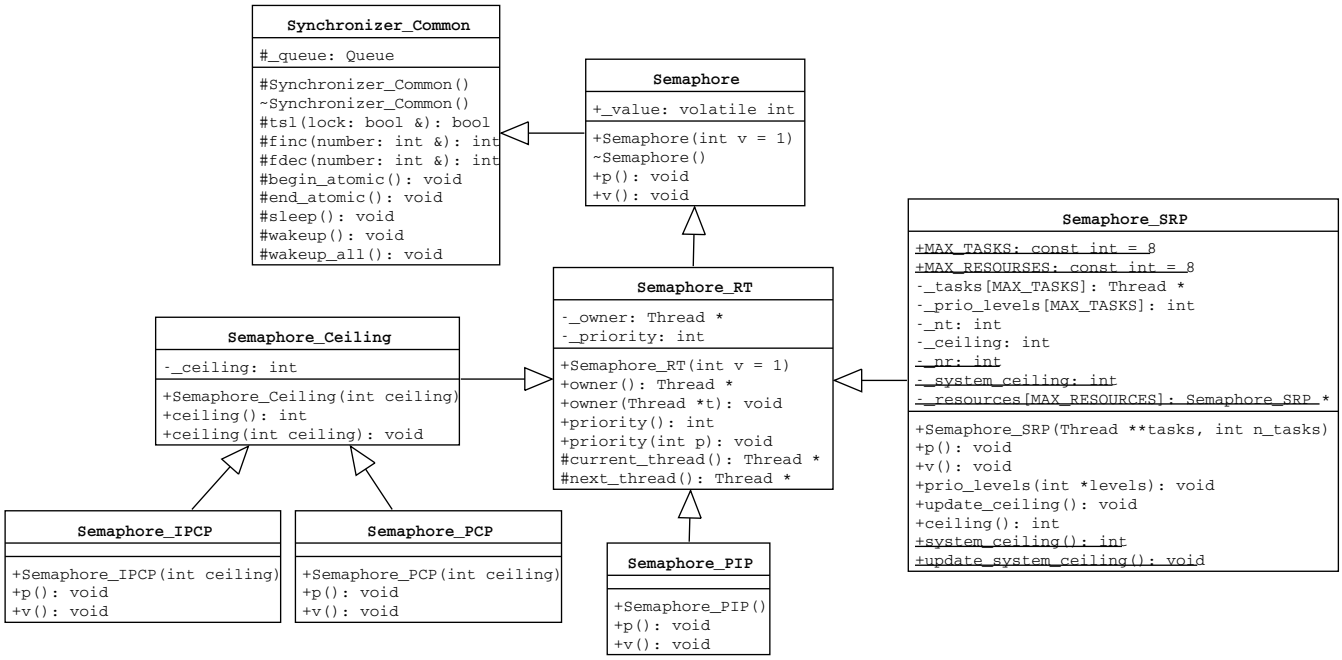

Figure 1. UML class diagram of the synchronization protocols.

are protected, which means that they are only accessible by subclasses.

The `Semaphore` class implements the traditional *p* and *v* semaphore operations [9]. The class has an integer (`_value`) as attribute, which is used to count the signals issued by the *p* and *v* (decrement and increment, respectively) through the `fdec` and `finc` operations implemented in the base class. It also uses the `sleep`, `wakeup`, and `wakeup_all` operations from the base class.

The `Semaphore_RT` class is common to all real-time resource access protocols. It has two attributes, `_owner` and `_priority` that represent, respectively, the current thread that owners the semaphore (*i.e,* a thread that has entered a critical section through the *p* operation) and the priority of that thread. The class offers public methods to set and get the attributes. Also, it has two protected methods, `current_thread` and `next_thread`, that return the current thread being executed (note that it can be different from the `_owner`) and the next thread that is the head of the semaphore's queue. These two operations are required by the PIP, PCP, and IPCP protocols.

The `Semaphore_PIP` class implements the priority inheritance behavior of the PIP protocol. It overwrites the *p* and *v* methods from the `Semaphore` to include the handling of the priority inheritance. For the *p* operation, the class first checks whether there is a thread inside the critical section by verifying if `_owner` is zero. If so, the calling thread becomes the new semaphore's owner. If not, there is a test to check whether the calling thread priority is greater than the owner's priority. If it is, owner has its priority raised to the calling thread's priority. Then, the *p* operation from the `Semaphore` is called to conclude the work. This is all done in an atomic state (*i.e.,* interrupts are disabled). For the *v* operation, if there is a semaphore owner, `Semaphore_PIP` checks if there is another thread waiting on the Semaphore's queue to enter the critical section. If so, the `_owner` and `_priority` are updated. If not, `_owner` and `_priority` receives zero as value. Then, the *v* operation from the `Semaphore` is called to conclude the work. The instruction within the *v* are also performed with interrupts disabled.

The `Semaphore_Ceiling` class is common to all ceiling-based protocols, such as IPCP and PCP. It adds an integer attribute (`_ceiling`), which represents the semaphore's ceiling. It also has the set and get methods for the attribute. The `Semaphore_PCP` class implements the PCP protocol by overwriting the *p* and *v* methods. The protocol raises the semaphore's owner thread priority to the ceiling whenever there is a call to *p*. If the semaphore has no owner, then the current thread becomes the new owner and the semaphore's priority is set to the thread's priority. After that, the *p* method of the `Semaphore` is called. The *v* operation reestablishes the owner thread's priority (in case it was raised) and checks whether there is another thread waiting on the semaphore's queue. Then, it calls the *p* method of the `Semaphore`.

The `Semaphore_IPCP` class implements the IPCP protocol, in a similar way to the `Semaphore_PCP` class. The only difference is that the owner priority is raised to the semaphore's ceiling whenever a thread enters the critical section. Also, the owner priority is always reestablished to its original priority whenever it calls *v*.

Finally, the `Semaphore_SRP` class implements the Stack Resource Policy (SRP) protocol. It is statically configured (at compile time) to support a given number of tasks per resource, and also a set number of resources in the whole system. Every instance has a resource ceiling attribute (`_ceiling`), which corresponds to the highest preemption level amongst the tasks that would block if accessing the resource. Whenever the

semaphore value changes, the resource recalculates its resource ceiling and also the system ceiling.

To make this dynamic accounting possible, the semaphore stores a list of tasks that access the resource, and the maximum number of resource units the task can hold at once (`_prio_levels` attribute). The system also needs to keep track of all existing SRP resources, and does so in a static array attribute (`_resources`).

A. Implementation in an RTOS

We have implemented the described protocols design in the Embedded Parallel Operating System (EPOS) [8, 10]. EPOS is a multi-platform, object-oriented, component-based, embedded system framework implemented in C++. EPOS is the first open-source RTOS designed from scratch that supports partitioned, global, and clustered versions of EDF, RM, LLF, and DM scheduling policies [11]. A complete review of the real-time support on EPOS can be found in [11]. We choose EPOS, because it supports static and dynamic scheduling and it is written in an object-oriented language. Thus, it is compatible with our proposed design. Moreover, EPOS until this work did not have a complete support for real-time resource access protocols. We believe that the proposed design can be replicated in any object-oriented RTOS written in C++ and that has EDF and RM schedulers.

Figure 2 shows the implementation of the `Semaphore_PCP` *p* operation. It uses the *begin_atomic* method from the `Synchronizer_Common` to disable interrupts. Then, it uses the methods from `Semaphore_RT` class to get the current running thread and modify or not the semaphore's ceiling. The `RT_Thread` class represents a running thread and it provides two *priority* methods to change and to get the real-time thread's priority. It is clear that code reuse is achieved by the inherent use of the class hierarchy. For instance, the `Semaphore` *p* operation is called through the `Semaphore_RT` class, in the operation number 15 in the sequence diagram class. The other protocols operations use the same strategy to allow maximum code re-usability.

IV. Experimental Evaluation

This section describes the experimental evaluation of the previously described implementation. Our objectives are threefold: (i) to measure the memory consumption of the implementation; (ii) to evaluate the run-time overhead of the implementation; and (iii) to verify the impact of the run-time overhead into the system schedulability. The next subsections show the obtained results for the memory consumption, run-time overhead, and schedulability analysis, respectively.

A. Memory Footprint

For measuring the memory consumption of our implementation, we have executed EPOS on top of the EPOSMote III platform [10]. EPOSMote III features an ARM Cortex-M3 32 MHz processor, with 32 Kb of RAM, and 512 Kb of flash. We used the *GNU gcc* compiler for ARM at version 4.4.4 to generate the code. For measuring the memory footprint, we used the *GNU objdump tool* available together with the gcc toolkit.

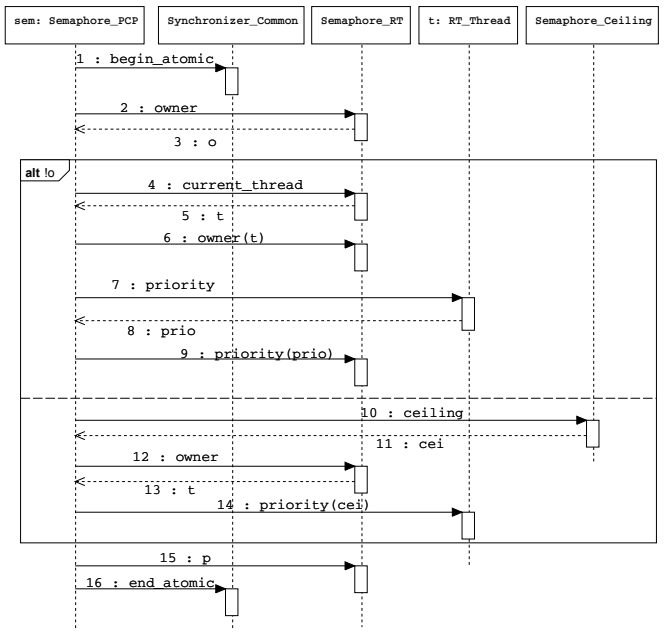

Figure 2. UML sequence diagram of the `Semaphore_PCP` *p* operation.

Table I shows the obtained memory footprint for each class (see Figure 1). The table also shows the total lines of code, just to correlate the memory footprint with the implementation. Memory usage is split into code, data, and static data sections. Data represents the memory consumed by an instance of the corresponding semaphore type, not including the footprint of the base classes. The total memory consumption describes the memory consumed by the implementation of the semaphore subclass and a single corresponding object instance. For instance, the total memory consumption of the `Semaphore_PIP` is 296 bytes, which represents its own 140 bytes summed with `Semaphore` and `Semaphore_RT` memory consumptions. It is important to highlight that code from the `Synchronizer_Common` class is not represented in the Table, because it is inlined into the methods that use the base class. It means that its code is implemented in the header file to improve the run-time overhead by avoiding explicitly method calls.

Table I. Memory footprint (in bytes) and lines of code of the implemented classes.

Class	Code	Data	Static Data	Total Mem. Consum.	Lines of Code	
					Header	Source
Semaphore	132	16	0	148	10	10
Sem. RT	0	8	0	8	24	0
Sem. Ceiling	0	4	0	4	9	0
PIP	140	0	0	140	8	35
PCP	144	0	0	144	8	28
IPCP	144	0	0	144	8	28
SRP	368	64	68	504	78	18

B. Run-time Overhead

For measuring the run-time overhead of the implementations, we used the EPOSMote's 32 MHz timer with ±40ppm accuracy. For each protocol, the test consisted of a set of 20 tasks that would try to acquire the same resource in a cascade, and subsequently release the resource. Then the relevant code

sections in the OS were instrumented to account for their execution time. Every *p* and *v* method of every semaphore type was instrumented, as well as the system code responsible for queuing and dequeuing tasks, which is used by the base semaphore implementation. These sections were timed with a small helper utility, that wraps the code, setups a hardware timer peripheral, and uses it to count the amount of time consumed in system clock cycles by the code section. We repeated the test 10 times and extract the worst-case run-time overhead from these executions.

Figure 3 presents the obtained worst-case run-time overhead for each protocol in µs to perform *p* and *v* operations, as a function of the number of tasks waiting on the semaphore. It was observed that for *p* operations of the PIP, IPCP, and PCP protocols, the overhead depends significantly on the number of tasks currently waiting on the semaphore. This is a consequence of queuing tasks when they block waiting on the resource. This queue is implemented as a linked list, and as the queue grows larger, the enqueuing overhead increases. This queue overhead overcomes the overhead of the protocols' code. However, we can note a very small difference when there is no tasks waiting on the semaphore. In that case, IPCP has a smaller overhead (almost imperceptible in the graph).

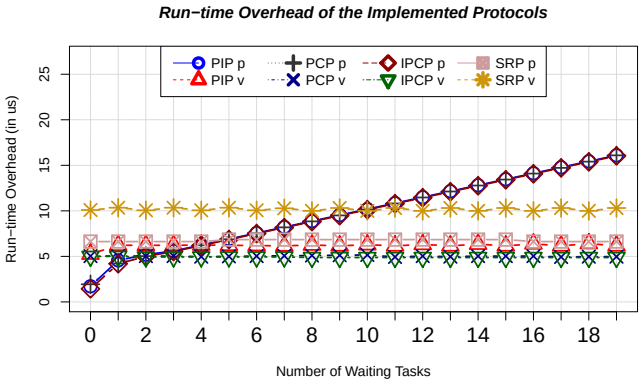

Figure 3. The obtained run-time overhead running the implemented protocols on the EPOSMote III platform.

In contrast, the overhead for the *v* operation does not change with the number of tasks in the queue, because dequeuing is always done from the head of the queue, so this is a constant-time operation. PCP and IPCP have almost the same overhead for the *v* operation, because both protocols do similar operations. The overhead for PIP, however, is a little bit larger (up to 1.2 µs) due to its more sophisticated operations. The *v* operation of the SRP is the worst, because it demands an updating of the system ceiling, which is performed in a loop (its size is equal to the number of tasks that use the resource, 20 in our experiments).

The case of tasks being blocked on a semaphore under SRP should never happen in principle. This is because of the preemption test, that disables a task from beginning execution if one of its resource accesses would cause it to block. However, in our practical implementation, non-real-time tasks do not have an assigned preemption level, and are unaffected by SRP. Therefore, in the case of a soft real-time application that makes mixed use of tasks types, tasks may still block on a resource.

C. Schedulability Impact

For measuring the impact of the run-time overhead into the schedulability of task sets, we used a methodology similar to the one proposed by Yang et al [3], adapting their methodology to our scenario (*i.e,* uniprocessor system). We generated several task sets according to the following rules. A task's period p_i was randomly chosen from a log-uniform distribution ranging over [10ms, 100ms] (*homogeneous periods*) or [1ms, 1000ms] (*heterogeneous periods*). Each task's utilization u_i was randomly chosen from a uniform distribution ranging over [0.05ms, 0.1ms] (*light utilizations*) or [0.1ms, 0.25ms] (*medium utilizations*), and the task's WCET e_i was set to $e_i = p_i \times u_i$. The number of shared resources was varied across $n_r \in \{1, 2, 4, 8\}$. For each resource, there is a probability p^{acc} associated with each task to access the respective resource. We vary the probability across $p^{acc} \in \{0.1, 0.25, 0.5\}$. The maximum critical section length $L_{i,q}$ was chosen uniformly from [1µs, 25µs] (*short*), [25µs, 100µs] (*medium*), and [100µs, 500µs] (*long*). Each task's job uses a resource only once per activation ($N_{i,q}$ = 1).

For each configuration (144 combinations[2]), we generated 1000 task sets per utilization cap, varying it from 0.1 to 1. For each task set and utilization cap, we first applied the traditional PIP, PCP, and SRP schedulability analyses [2] without considering any run-time overhead. IPCP has the same schedulability test as PCP [2]. For PIP, PCP, and IPCP we consider the RM scheduling, while for SRP we consider the EDF scheduling, to evaluate both static- and dynamic-priority scheduling approaches. Then, we inflated each task's WCET e_i with the obtained run-time overhead (see Figure 3). To obtain the correct run-time overhead, we used the following approach. Considering a task τ_i and a task set with n tasks. For each resource R_j used by τ_i, we iterated over the tasks finding the other tasks that also use R_j. Then, we obtained the maximum number of tasks that share the same resources with τ_i. This gave us the worst-case number of tasks that can wait on the same resource. We used this number to take the overhead. We implemented the task generation methodology as well as the protocol schedulability analyses [2] in the SchedCAT tool [12][3].

Due to the large combination of configurations, we only focus here on the major findings. Figures 4–9 shows the representative graphs. On the *x-axis* we vary the utilization cap of the generated task sets. On the *y-axis* we vary the schedulability ratio. For instance, a schedulability ratio of 0.6 means that 60% out of the 1000 task sets were schedulable. Below, we discuss the main observed behaviors from the obtained results.

Efficient implementation provides schedulability bounds close to the theoretical ones. Our results indicate that proper implementation of resource access protocols provide small impact on the schedulability ratio of task sets in most cases. For instance, in a scenario with high demand for shared resources (Figures 7–9), the difference between the schedulability ratio with and without overhead is less than 5% in all scenarios. An exception is when the utilization cap

[2]Due to space constraint, we only show the graphs of 6 combinations in the paper. All graphs will be available at http://epos.lisha.ufsc.br.

[3]The scripts used during the experiments are available online at http://epos.lisha.ufsc.br.

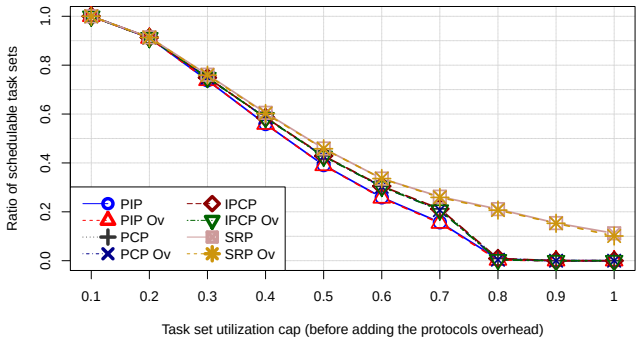

Figure 4. Heterogeneous periods, light utilizations, long critical sections, $n_r = 4$, $p^{acc} = 0.25$, and $N = 1$.

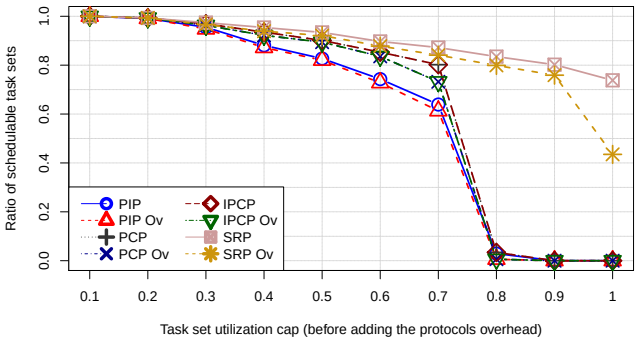

Figure 5. Heterogeneous periods, light utilizations, medium critical sections, $n_r = 4$, $p^{acc} = 0.25$, and , $N = 1$.

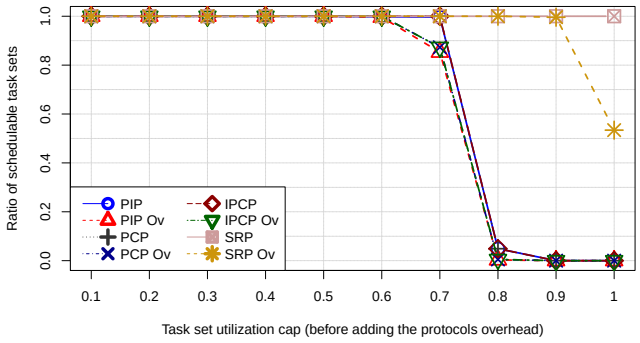

Figure 6. Heterogeneous periods, light utilizations, short critical sections, $n_r = 4$, $p^{acc} = 0.25$, and , $N = 1$.

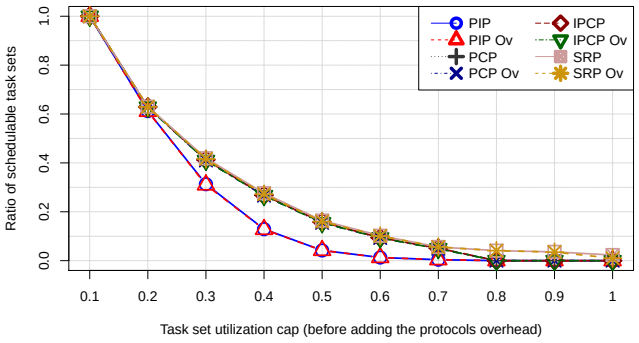

Figure 7. Heterogeneous periods, light utilizations, long critical sections, $n_r = 8$, $p^{acc} = 0.5$, and , $N = 1$.

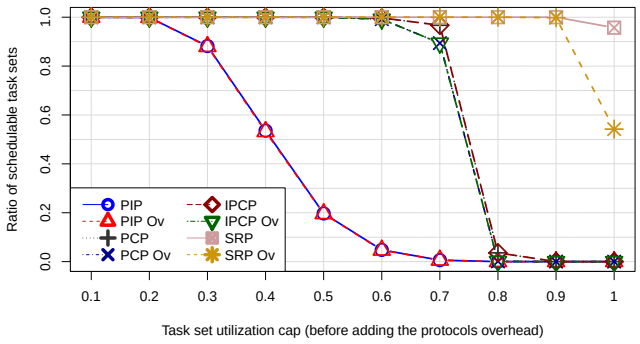

Figure 8. Homogeneous periods, light utilizations, long critical sections, $n_r = 8$, $p^{acc} = 0.5$, and , $N = 1$.

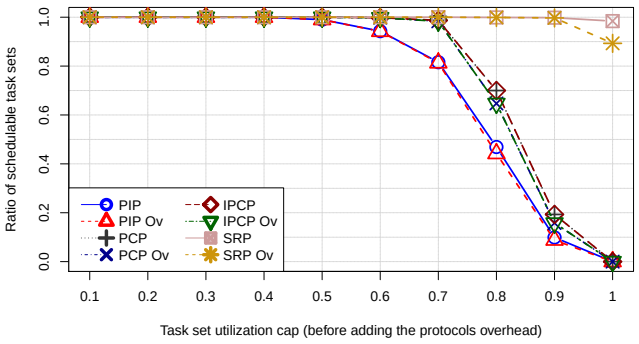

Figure 9. Homogeneous periods, medium utilizations, long critical sections, $n_r = 8$, $p^{acc} = 0.5$, and $N = 1$.

is between 0.9 and 1, where the overhead has an important influence on the schedulability, because the utilization of the task sets are very close to the scheduling bounds (see Figure 5 for example).

Critical section sizes heavily impact schedulability for light-utilization task sets. As expected, Figures 4, 5, and 6 show that for task sets with a great number of low utilization tasks, critical section sizes have a strong impact on the schedulability ratio. We can note that with longer critical sections, the schedulability is reduced. This is a consequence of blocking time bounds for all protocols being derived directly from critical section lengths.

Homogeneity of task periods has a major effect on the schedulability ratio. For a given total utilization, heterogeneous task sets contain tasks with shorter periods. Since critical section lengths for a given resource are the same for all tasks, short period tasks were more affected by blocking-time bound terms on the schedulability analyses. Because of this, the schedulability ratios for task sets with heterogeneous periods were lower (see Figures 7 and 8 for instance).

Schedulability is higher in task sets with medium utilizations. This is mainly due to the fact that task sets with medium utilizations have fewer tasks when compared to light utilizations (compare Figures 8 and 9). It can also be observed in Figure 8 that for a higher number of tasks, if periods are homogeneous, PCP performs much better than PIP.

Probability of accessing a resource impacts the schedulability of task sets considering overhead. When we used higher probabilities to access shared resources, the schedulability of task sets considering the run-time overhead decreased, as expected. This behavior can be clearly seen in Figures 4 and 7.

Ceiling-based protocols are better than the PIP. In all tests, the ceiling-based protocols, PCP and IPCP, have presented better schedulability ratios than PIP. This is also expected, as the PCP/IPCP have lower blocking times given by their schedulability test when compared to the PIP test. PIP has shown to be a little bit less sensitive to the run-time overhead than PCP and IPCP.

V. Related Work

Suspension-based resource access protocols for uniprocessor real-time systems were first proposed by Sha et al [6]. The authors have proposed PCP and PIP and the work has served as basis to many other researches. SRP was proposed by Baker in 1991 [7] and it was also a seminal work, mainly for providing resource access protection for dynamic scheduling.

Several general-purposed OSes and RTOSes implement some of the analyzed real-time resource access protocol. For instance, FreeRTOS employs PIP in the mutex primitive [13] and also supports SRP [14]. $LITMUS^{RT}$ supports PCP, SRP, and several multiprocessor resource access protocols [15, 16]. The L4 microkernel [17] and Linux support priority inheritance. Linux also implements IPCP, under the name PRIO_PROTECT in the pthreads library. Lee and Kim implemented PIP in the µC/OS-II kernel and measured the run-time overhead running the implementation on top of the CalmRISC16 evaluation board [18]. The observed run-time overhead for the *p* and *v* semaphore operations was 30.5 µs. Researchers also proposed to move the mechanisms to control the priority inheritance [19] or the semaphore structures [20] to the hardware in order to reduce the run-time overhead.

Wang et al. implemented multi-resource versions of PIP and PCP in a component-based OS for controlling the access to shared stacks [21]. In their experimental evaluation considering the schedulability of generated tasks, PIP has performed better than PCP. Thus, they have concluded that PIP has potential to provide a high-degree of schedulability [21]. In our experimental evaluation, however, PIP has presented similar performance in terms of overhead when compared to PCP, but had worse schedulability ratios.

Although not directly involved with this paper, resource access protocols for multiprocessors have been the subject for many researchers recently. MPCP [22] and MrsP [23] are two multiprocessor variants of the PCP. Flexible Multiprocessor Locking Protocols (FMLP) is a collection of protocols for global and partitioned scheduling [24]. It was designed to efficiently deal with short non-nested access and to allow unrestricted critical section nesting. Parallel Priority Ceiling Protocol (P-PCP) extends PIP to avoid unfavorable blocking situations, but increases the run-time overhead [25]. Biondi and Brandenburg [26] have revisited four synchronization protocols under partitioned EDF (P-EDF) scheduling and compared them in terms of schedulability. They concluded that the lock-free synchronization approach offers advantages on asymmetric multiprocessing platforms. Yang et al. proposed new schedulability analyses based on linear programming for several multiprocessor semaphore-based locking approaches [3]. The authors claim that the new analyses are more accurate than prior approaches.

VI. Conclusion

This paper presented an object-oriented design of four uniprocessor real-time resource protocols (PIP, PCP, IPCP, and SRP). We have implemented the proposed design in an RTOS and measured the memory footprint and run-time overhead of the implementation. By compiling and running our implementation in a microcontrolled platform, we proved that it allows small memory footprint (from 296 to 660 bytes, depending on the protocol) and run-time overhead (less than 20 µs for 20 tasks). Moreover, we used the run-time overhead to analyze how it affects the system schedulability and proved that efficient RTOS implementation of uniprocessor resource access protocols has few impact on the system schedulability.

As future work, we plan to extend the design to support multiprocessor resource access protocols, such as MPCP and MSRP, then implement these protocols in EPOS and repeat the schedulability analysis considering the run-time overhead.

References

[1] J. Liu, *Real-Time Systems*, 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000.

[2] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*, 3rd ed. Springer Publishing Company, Incorporated, 2011.

[3] M. Yang, A. Wieder, and B. B. Brandenburg, "Global real-time semaphore protocols: A survey, unified analysis, and comparison," in *Real-Time Systems Symposium, 2015 IEEE*, Dec 2015, pp. 1–12.

[4] M. Jones. (1997, Dec) What really happened on mars? [Online]. Available: http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Mars_Pathfinder.html

[5] G. E. Reeves. (1997, Dec) What really happened on mars? [Online]. Available: http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Authoritative_Account.html

[6] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, Sep 1990.

[7] T. P. Baker, "Stack-based scheduling for realtime processes," *Real-Time Syst.*, vol. 3, no. 1, pp. 67–99, Apr. 1991.

[8] A. A. Fröhlich, *Application-Oriented Operating Systems*, ser. GMD Research Series. Sankt Augustin: GMD - Forschungszentrum Informationstechnik, Aug. 2001, no. 17.

[9] E. W. Dijkstra, "Cooperating sequential processes," in *Programming Languages: NATO Advanced Study Institute*, F. Genuys, Ed. Academic Press, 1968, pp. 43–112.

[10] EPOS. (2016, Jul) Website. [Online]. Available: http://epos.lisha.ufsc.br

[11] G. Gracioli, A. A. Fröhlich, R. Pellizzoni, and S. Fischmeister, "Implementation and evaluation of global and partitioned scheduling in a real-time OS," *Real-Time Systems*, vol. 49, no. 6, 2013.

[12] (2016, Jul) Schedcat: Schedulability test collection and toolkit. [Online]. Available: http://www.mpi-sws.org/bbb/projects/schedcat

[13] (2016, Jul) Freertos web-site. [Online]. Available: http://www.freertos.org/

[14] R. Inam, J. Mäki-Turja, M. Sjödin, and M. Behnam, "Hard real-time support for hierarchical scheduling in freertos," in *7th annual workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 11)*, July 2011, pp. 51–60.

[15] B. B. Brandenburg and J. H. Anderson, "An implementation of the pcp, srp, d-pcp, m-pcp, and fmlp real-time synchronization protocols in litmusrt," in *Proceedings of the 2008 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, ser. RTCSA '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 185–194.

[16] R. Spliet, M. Vanga, B. B. Brandenburg, and S. Dziadek, "Fast on average, predictable in the worst case: Exploring real-time futexes in litmusrt," in *Real-Time Systems Symposium (RTSS), 2014 IEEE*, Dec 2014, pp. 96–105.

[17] J. Liedtke, "On micro-kernel construction," in *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, ser. SOSP '95. New York, NY, USA: ACM, 1995, pp. 237–250.

[18] J.-H. Lee and H.-N. Kim, "Implementing priority inheritance semaphore on uc/os real-time kernel," in *IEEE Workshop on Software Technologies for Future Embedded Systems, 2003*, May 2003, pp. 83–86.

[19] B. E. S. Akgul, V. J. M. III, H. Thane, and P. Kuacharoen, "Hardware support for priority inheritance," in *24th IEEE Real-Time Systems Symposium, 2003. RTSS 2003*, Dec 2003, pp. 246–255.

[20] H. Marcondes and A. A. Fröhlich, *A Hybrid Hardware and Software Component Architecture for Embedded System Design*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 259–270.

[21] Q. Wang, J. Song, and G. Parmer, "Execution stack management for hard real-time computation in a component-based os," in *Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd*, Nov 2011, pp. 78–89.

[22] R. Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," in *Proceedings of the 10th International Conference on Distributed Computing Systems, 1990*, May 1990, pp. 116–123.

[23] A. Burns and A. J. Wellings, "A schedulability compatible multiprocessor resource sharing protocol – mrsp," in *2013 25th Euromicro Conference on Real-Time Systems*, July 2013, pp. 282–291.

[24] A. Block, H. Leontyev, B. B. Brandenburg, and J. H. Anderson, "A flexible real-time locking protocol for multiprocessors," in *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*, Aug 2007, pp. 47–56.

[25] A. Easwaran and B. Andersson, "Resource sharing in global fixed-priority preemptive multiprocessor scheduling," in *30th IEEE Real-Time Systems Symposium, 2009, RTSS 2009.*, Dec 2009, pp. 377–386.

[26] A. Biondi and B. B. Brandenburg, "Lightweight real-time synchronization under p-edf on symmetric and asymmetric multiprocessors," in *Proc. of the ECRTS 2016*, 2016, pp. 1–11.