A comparison of MPCP and MSRP when sharing resources in the Janus multiple-processor on a chip platform

Paolo Gai, Marco Di Natale, Giuseppe Lipari,
Scuola Superiore Sant'Anna, Pisa, Italy
{pj,marco,lipari}@sssup.it

Alberto Ferrari
PARADES, Roma, Italy
aferrari@parades.rm.cnr.it

Claudio Gabellini, Paolo Marceca
Magneti Marelli Powertrain Div., Bologna, Italy
{marceca,gabellini}@bologna.marelli.it

Abstract

The new generation of embedded systems for automotive applications can take advantage of low-cost multiprocessor system-on a chip architectures. The real-time software applications running on these systems require real-time processor scheduling, and also require the management of the communication and synchronization of tasks executing on different processors with limited blocking time. Conventional real-time technologies, like the Rate Monotonic scheduling algorithm together with the Multiprocessor Priority Ceiling Protocol (MPCP) can be used to this purpose. In earlier work, we proposed the Multiprocessor Stack Resource Policy (MSRP) for scheduling tasks and sharing resources in multiprocessor on a chip architectures. In this paper we present an experimental evaluation that compares the performance of our algorithm with a solution based on Rate Monotonic and MPCP in the context of the Janus multiple processor architecture. The evaluation of the algorithm has been triggered by our ongoing research in the automotive domain. We report on two sets of experiments: the first addresses a range of generic task configurations to see if one of the algorithms can clearly outperform the other. The results show MSRP to be better for random task periods but are probably not conclusive. Later, we focus on a more application-specific (also more restrictive) architecture design representing a typical automotive application: a power-train controller. In this case, MSRP clearly performs better. The performance gap between the two policies can be further increased when considering that MSRP is much simpler to implement, it has a lower overhead, and it allows RAM memory optimization.

Keywords: real-time, operating systems, multiprocessor, scheduling, system-on-a-chip

1 Introduction

The exponential growth of silicon capacity allows the proliferation of embedded systems in different application domains offering unprecedented performance and functionality. In present and future hard real-time automotive applications, the introduction of multiple-processor-on-a-chip architectures is seen as a very likely solution [4].

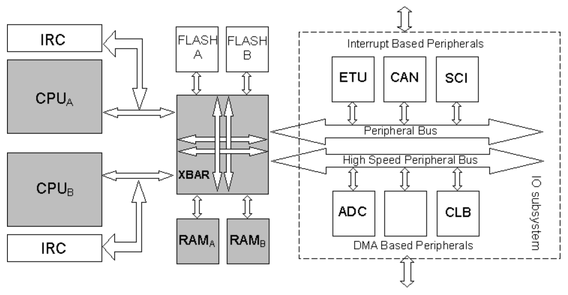

Figure 1. The Janus Dual Processor system

The Janus microntroller (Figure 1), developed by PARADES, ST Microelectronics and Magneti Marelli in the context of the MADESS[1] project, is an example of a dual-processor platform for power-train applications. Two 32-bit ARM7TDMI processors connected by a crossbar switch to 4 memory banks and two peripheral buses for I/O processing (low and high bandwidth) provide twofold computational power, compared to a single (ARM7TDMI) processor architecture, at very low increment of the silicon area, i.e. at comparable system costs. Both CPUs share the same address space. The main memory is organized in different modules and types: SRAM and FLASH. In architectures with multiple processors, memory access is the most important bottleneck of the system. Almost any communication flow is between the memory and other system components. To allow correct synchronizations and communication among tasks allocated to different processors, the architecture provides hardware support for inter-processor communication by interrupt inter-processor mechanisms and for shared memory by atomic test-and-set.

The applications running on the new single-chip platform require predictable (and fast) scheduling algorithms.

[1]`http://www.madess.cnr.it/Summary.htm`

In addition, kernels must fit in a few kbytes of memory, and, together with the application, they must use the smallest possible amount of RAM memory. Resource sharing must be carefully handled and all communication primitives on shared memory must be designed in order to allow for a limited blocking time.

In previous work [5] we presented the MSRP scheduling and resource sharing protocol, which allows reducing the amount of RAM memory allocated to the task stacks. One of the possible drawbacks of the MSRP policy (and a major concern for the automotive application developers that cooperate with us in MADESS) is the cost of spin locking in multiprocessor real-time systems when compared to other policies. In contrast, the multiprocessor priority ceiling protocol or MPCP, probably the best known policy for bounding blocking time in a predictably way in multiprocessor systems, avoids spin-locking, but does not allow sharing the stack space of tasks. Furthermore, it requires a non trivial run-time support, which results in greater overhead when compared to the implementation of MSRP. In order to settle this dispute, we performed experiments comparing MSRP with MPCP in Janus. The experiments are in two stages. In the first stage, our simulator evaluates the schedulability of a number of generic task sets to see if one of the algorithms can clearly outperform the other. The results are not conclusive, except that (as expected) MSRP is better when considering few global resources and short critical sections. In the second stage we focus on a domain-specific example: a task set implementing a power-train controller, which is the representative of a typical automotive application and our target demonstrator in MADESS. For this case, MSRP clearly outperforms MPCP, proving the viability of a spin-lock based approach for sharing resources on the Janus platform.

The structure of the paper is the following: Sections 2 and 3 contain the description of our terminology and references to fundamental work in this area. Sections 4 and 5 contain a short introduction to the MPCP and MSRP algorithms with the corresponding schedulability analysis formulas. Section 6 contains a comparison study of the two methods, discussing their implementation. Section 7 provides a general introduction to power-train control systems and introduces the thread architecture of the case study. Finally, Section 8 contains the experimental results of our simulations in the general case and in our target automotive application.

2 Assumptions and terminology

In the paper, we use the terms thread and task interchangeably. The assumptions and definitions for the terms and symbols used in the paper are the following:

Our embedded application consists of a set $\mathcal{T} = \{\tau_1, \tau_2, \dots, \tau_n\}$ of real time tasks to be executed on a set $\mathcal{P} = \{P_1, \dots, P_m\}$ of processors. The subset of tasks assigned to processor P_k will be denoted by $T_{P_k} \subset \mathcal{T}$.

A task τ_i is a infinite sequence of jobs (or instances) $J_{i,j}$. Every job is characterized by a release time $r_{i,j}$, an execution time $c_{i,j}$, an absolute deadline $d_{i,j}$ and a priority p_i

A task can be periodic or sporadic. Without loss of generality, we use the same symbol θ_i to indicate the period or the minimum interarrival time of task τ_i. In the following a task will be characterized by a worst case execution time $C_i = \max\{c_{i,j}\}$ and a period θ_i. We assume that the relative deadline of a task is equal to its period θ_i: thus, $d_{i,j} = r_{i,j} + \theta_i$.

Tasks are allowed to access mutually exclusive resources through critical sections. Let $\mathcal{R} = \{\rho^1, \dots, \rho^p\}$ be the set of shared resources. The k–th critical section of task τ_i on resource ρ^j is denoted by ξ^j_{ik} and its maximum duration is denoted by ω^j_{ik}.

Finally, we suppose that tasks have been statically allocated to Processors and are always executed on the same processor. Depending on this allocation, resources can be divided in *local* and *global* resources. A critical section protecting a global resource is called *global critical section* or *gcs*.

3 Related Work

The **uniprocessor Priority Ceiling Protocol or PCP** (see [12]) is one of the best known policies for avoiding priority inversions and limiting the blocking time in single processor systems. The PCP policy defines a (static[2]) ceiling attached to each semaphore (resource ρ^k) as the maximum priority among all tasks that can possibly lock the semaphore.

$$\text{ceil}(\rho^k) = \max_i\{p_i \mid \tau_i \text{ uses } \rho^k\}.$$

and a dynamic *system ceiling* is defined as

$$\Pi_s(t) = \max[\{ceil(\rho^k) \mid \rho^k \text{ is currently locked}\} \cup \{0\}].$$

Job J_i requesting a resource is blocked if its priority is not higher than the system ceiling. The Priority Ceiling Protocol, which can be used together with the Rate Monotonic (RM) scheduler [8], ensures that a job can be blocked only once when accessing a shared resource held by a lower priority job. This delay is called *blocking time* and denoted by B_i. The maximum local blocking time for each task τ_i can be calculated as

$$B_i = \max_{\tau_j \in \mathcal{T}, \forall h}\{\omega^k_{jh} \mid p_i > p_j \wedge p_i \le \text{ceil}(\rho^k)\}. \quad (1)$$

The schedulability condition for the PCP protocol when used together with a RM scheduler is:

$$\forall i, \ \ 1 \le i \le n \quad \sum_{k=1}^{n} \frac{C_k}{\theta_k} + \frac{B_i}{\theta_i} \le n(2^{1/n} - 1) \quad (2)$$

[2]In the case of multi-units resources, the ceiling of each resource is dynamic as it depends on the current number of free units.

The Stack Resource Policy was proposed by Baker in [2] for scheduling a set of real-time tasks on a uniprocessor system. The SRP is similar to the Priority Ceiling Protocol, but it has the additional property that a task is never blocked once it starts executing. Like PCP it can be used together with RM or EDF.

According to the SRP, every real-time (periodic and sporadic) task τ_i must be assigned a priority p_i and a static preemption level λ_i, such that: *task τ_i is not allowed to preempt task τ_j, unless $\lambda_i > \lambda_j$.*

The definition of the semaphore ceiling used by SRP is slightly different from the one used by PCP, since it does involve preemption levels instead of priorities:

$$\text{ceil}(\rho^k) = \max_i\{\lambda_i \mid \tau_i \text{ uses } \rho^k\}.$$

and the SRP scheduling rule states that: *"a job is not allowed to start executing until its priority is the highest among the active jobs and its preemption level is greater than the system ceiling"*.

The SRP ensures that, once a job is started, it cannot be blocked until completion; it can only be preempted by higher priority jobs. However, the execution of a job $J_{i,k}$ with the highest priority in the system could still be delayed by a lower priority job. In [2] Baker proved that Formula 2 can be used for checking the schedulability of tasks under SRP+RM (or EDF). The maximum blocking time can be computed in exactly the same way it is computed for PCP, with the only exception that preemption levels must be considered instead of priorities.

From an implementation viewpoint, SRP allows tasks to share a unique stack. In fact, a task never blocks its execution. The implementation of the SRP is straightforward as there is no need to implement waiting queues. Furthermore, by disabling (some) preemption [11], the requirements for stack space can be reduced. Our MSRP algorithm extends this idea to dynamic scheduling and multiprocessor systems.

The *Multiprocessor Priority Ceiling Protocol* (MPCP) has been proposed by Rajkumar in [10] for scheduling a set of real-time tasks with shared resource on a multiprocessor. It extends the Priority Ceiling Protocol [12] for global resources. Since this policy is the term of comparison for our MSRP policy we will spend some extra time discussing its features.

4 The MPCP Multiprocessor Priority Ceiling Protocol

If tasks block on semaphores protecting global resources, the concept of blocking needs to include also *remote blocking* (when a job has to wait for the execution of a task of any priority assigned to another processor.) MPCP extends the priority ceiling protocol to multiprocessor systems with the assumption that tasks are statically bound to processors and scheduled according to the rate monotonic policy.

The goal of MPCP is to bound the remote blocking duration of a job as a function of the duration of critical sections of other jobs and not as a function of the duration of non-critical code. As a direct consequence, it is necessary that global critical sections are assigned a ceiling that is higher than the priority of any other task in the system. If p_H is the highest priority among all tasks, a priority of $p_H + 1 + \max_i\{p_i \mid \tau_i \text{ uses } \rho^k\}$ is the priority ceiling for the semaphore protecting the global resource ρ_k. Other important design choices of MPCP are the following:

- jobs are suspended when they try to access a locked gcs;
- when a higher priority task is blocked on a global critical section local tasks can be executed and may even try a lock on local or global critical sections;
- when a global resource is released the task waiting on top of the semaphore list is awakened and inherits the priority of the global critical section.

One very important consequence of letting lower priority local tasks execute and possibly inherit the priority of global critical sections is the possibility of priority inversion occurring while a high priority task is blocked on a gcs. Other assumptions are the following: local critical sections do not make nested access to global resources and vice versa, furthermore, nested accesses to global critical sections are prohibited.

MPCP allows for a bounded blocking time and a formula exists for checking the schedulability of real-time tasks. The formula is an adaptation of Formula 2 (to be evaluated for each processor) with the only difference that the blocking factor B_i must account for local and global priority inversions. In order to simplify the formulation of the five factors that add up to form the factor B_i the following additional definitions are introduced.

Task τ_i can access local (i.e. allocated on the same processor) or global resources. The number of global critical sections executed by τ_i is n_i^G. $NL_{i,j}$ is the number of jobs with a lower priority than J_i on its processor. $\{J'_{(i)_r}\}$ is the set of jobs on processor P_r with *gcs* having priority higher than global critical sections that can directly block J_i. $NH_{i,r,j}$ is the number of global critical sections of job $J_j \in \{J'_{(i)_r}\}$ with higher priority than a global critical section on processor P_r, which can directly block J_i. $\{nGS_{i,j}\}$ is the set of global semaphores locked by both J_i and J_j. Finally, $NC_{i,j}$ is the number of global critical sections entered by J_j and guarded by elements of $\{nGS_{i,j}\}$.

The blocking time for a job J_i on processor P_j consists of up to five different factors:

$$B_i = B_{i_1} + B_{i_2} + B_{i_3} + B_{i_4} + B_{i_5}$$

where

- $B_{i_1} = n_i^G \omega_i^{local}$ (where ω_i^{local} is the longest critical section accessed by jobs with a priority lower than J_i executing on the same processor), since each time J_i needs a global semaphore may suspend, letting lower priority jobs execute on its processor. These low priority jobs can lock local semaphores and block J_i when it resumes its execution.

- $B_{i_2} = n_i^G \omega_j^{global}$ (where ω_j^{global} is the longest global critical section accessed by jobs with a priority lower than J_i executing on other processor) when job J_i tries to access a global critical section and finds it is accessed by a lower priority job on another processor.

- $B_{i_3} = NC_{i,j} \lceil T_i/T_j \rceil \omega_j^{global}$ for each higher priority job executing on a processor different from P_i and requesting the same global semaphore as J_i.

- $B_{i_4} = NH_{i,r,j} \lceil T_i/T_j \rceil \omega_j^{global}$ for higher priority global critical sections, which can preempt the global critical sections of lower priority jobs directly blocking J_i.

- $B_{i_5} = min(n_i^G + 1, n_k^G)\omega_j^{global}$ each time J_i tries to access a global critical section it can suspend letting lower priority jobs execute on its processor. These jobs can lock or queue on global semaphores and eventually execute at a priority higher than P_i and preempt it when it executes non global code.

5 Multiprocessor SRP

The MSRP policy provides a solution to the resource sharing as well as the task allocation problem. Since EDF+SRP cannot be directly applied to multiprocessor systems, we proposed an extension of these protocols. This solution allows tasks to use local resources under the SRP policy and to access global resources with a predictable blocking time without interfering with the local execution order. This mechanism, when used in conjunction with preemption thresholds, and the creation of non–preemptive task groups [11] [5] allows to perform time guarantees minimizing the requirement for RAM space. According to our MSRP policy, if a task tries to access a global resource and the resource is already locked by some other task on another processor, the task performs a busy wait (also called *spin lock*). The *spin lock time* should be reduced as much as possible (the resource should be freed as soon as possible). For this reason, the tasks become non-preemptable when executing a critical section on a global resource. The MSRP algorithm works as follows:

- For local resources, the algorithm is the same as the SRP algorithm. Tasks are allowed to access local resource through nested critical sections (it is not possible to nest global critical sections).

- When a task τ_i, allocated to processor P_k accesses a global resource ρ^j, the task becomes non–preemptable. Then, the task checks if the resource is free: in this case, it locks the resource and executes the critical section. Otherwise, the task is inserted in a FCFS queue on the global resource, and then performs a busy wait.

- When a task τ_i, allocated to processor P_k, releases a global resource ρ^j, the algorithm checks the corresponding FCFS queue, and, in case some other task τ_j is waiting, it grants access to the resource, otherwise the resource is unlocked. The task becomes again preemptable.

The spin lock time that every task allocated to processor P_k needs to spend for accessing a global resource $\rho^j \in \mathcal{R}$ is bounded from above by:

$$spin(\rho^j,\, P_k) = \sum_{p \in \{\mathcal{P} - P_k\}} \max_{\tau_i \in T_p,\, \forall\, h} \omega_{ih}^j.$$

Basically, the spin lock time increments the duration ω_{ih}^j of every global critical section ξ_{ih}^j, and, consequently, the worst case execution time C_i of τ_i. Moreover, it also increments the blocking time of the tasks allocated to the same processor with a preemption level greater than λ_i. We define the *actual worst case computation time* C_i' for task τ_i as the worst case computation time plus the total spin lock time:

$$C_i' = C_i + \sum_{\xi_{ih}^j} spin(\rho^j,\, P_k)$$

MSRP maintains the same basic properties of the SRP, that is, once a job starts executing it cannot be blocked, but only preempted by higher priority jobs and a job can experience a blocking time at most equal to the duration of one critical section (plus the spin lock time, if the resource is global) of a task with lower preemption level.

It is noteworthy that the execution of all the tasks allocated on a processor is perfectly nested (because once a task starts executing it cannot be blocked), therefore all tasks can share the same stack.

The blocking time for a task can be divided into blocking time due to local and global resources.

$$B_i = max(B_i^{local},\, B_i^{global})$$

where B_i^{local} and B_i^{global} are defined as:

$$B_i^{local} = \max_{j,h,k}\{\omega_{jh}^k \mid (\tau_j \in T_{P_i}) \wedge (\rho^k \text{ is local to } P_i) \wedge (\lambda_i > \lambda_j) \wedge (\lambda_i \le ceil(\rho^k))\}$$

$$B_i^{global} = \max_{j,h,k}\{\omega_{jh}^k + spin(\rho^k, P_i) \mid (\tau_j \in T_{P_i}) \wedge (\rho^k \text{ is global}) \wedge (\lambda_i > \lambda_j)\}$$

Suppose that tasks on processor P_k are ordered by decreasing preemption level. The schedulability test is as follows:

$$\forall\, P_k \in \mathcal{P} \qquad T_{P_k} = \{\tau_1, \ldots, \tau_{n_k}\} \quad \forall i = 1, \ldots, n_k$$
$$\sum_{l=1}^{i} \frac{C_l'}{\theta_l} + \frac{B_i}{\theta_i} \le 1 \qquad (3)$$

6 Comparing MSRP with MPCP

The blocking factors $B_{i_1}, \dots B_{i_5}$ of MPCP are the result of a worst case analysis and can be reduced by carefully examining the task set at hand. Nonetheless the guarantee formula is clearly extremely complicated. Consider also that PCP requires keeping track of local and global priority ceilings and the previous formula holds if the period enforcing technique (described in [10]) is used.

If, on the other hand, MSRP is used, we can expect to waste more local processor time due to the use of spin locks when trying to lock global resources. The guarantee formula of MSRP is simpler since we do not have to account for the events that cause the blocking factors B_{i_1} and B_{i_5} which are the consequence of suspending a task when trying to access a locked critical section.

At first sight, it would appear that, whenever global critical sections are sufficiently short, the MSRP approach would perform better (besides being much simpler to implement). On the other hand, MPCP should be better when global critical sections grow larger. We performed a first set of experiments trying to determine where this boundary lies and in what conditions should designers expect MSRP to perform adequately. Following the results of these experiments, we focused our analysis on a power-train application: a typical case study from the automotive domain.

6.1 Implementation notes

An implementation of the MPCP protocol can be basically divided in two parts [10]: the implementation of a local priority ceiling protocol and the implementation of the global inter–processor synchronization.

The local part of the protocol implementation can easily be done using a priority ordered Task queue, where the highest priority task in the queue is the running task. Moreover, a list of locked semaphores (ordered by ceiling) has to be maintained to allow the implementation of the inheritance of the priority.

The global part of the protocol subsumes the existence of a shared data structure that records the state of a global mutex. In particular, an ordered queue of the tasks that are blocked on the global resource has to be implemented. The low-level access to that data structure has to be done in mutual exclusion, and that is usually done using a spin-lock approach (the duration of the spin lock is not accounted into the guarantee equations, since it is bounded by the maximum time needed to handle the internal data structure) or using an inter-processor interrupt. Moreover, to guarantee a bounded blocking time the Period Enforcer technique must be implemented.

When using SRP there is no need to implement semaphores and queues for blocked tasks, and the blocking time experienced by each task can be predictably bound. Furthermore, the SRP allows multiple tasks to share a single stack. For these reasons, the SRP can be implemented with a small overhead and memory occupation. The implementation of MSRP on the Janus platform has been simplified by taking advantage of the fact that there are only two processors contending for the use of global resources. In particular, only one processor at a time can be blocked on a global resource, so FCFS queues are not needed for waiting tasks. Moreover, implementing a spin-lock mechanism on Janus only requires a negligible amount of code. Since all memory is shared between the two processors, a single memory location can be used to synchronize all tasks using the `swpb` ARM instruction.

7 The Power-train Control Application

7.1 Introduction

The goal of power-train control systems is to offer appropriate driving performance (e.g. driveability, comfort, safety) while minimizing fuel consumption and pollutant emissions. In an engine management system, the fuel injection and air intake are controlled to produce the desired mix to be transformed, by the combustion process, in torque and emissions. The combustion process takes place in the cylinders and the starting time is controlled by the sparks generated from the spark plugs. The produced torque is then applied to the power-train, which is controlled by the gear selection and clutch position. The final result is the force applied, through the wheels, to the entire vehicle. Driveability is an informal measure of how favorably this force is perceived by the driver under his/her action. The control strategy goal is achieved by means of several control inputs such as throttle position, fuel injection, spark ignition, gear selection and clutch position. Fuel injection, spark ignition and part of the gear-box control are angle-based, i.e. they must be synchronized with the *engine position*[3] *or drive-line angle*. The other control variables do not have these synchronization constraints and are called time based. To compute the engine position, the engine has two sensors (the crankshaft and cam-shaft toothed wheel sensors) providing two angular references used for injection and ignition synchronization. Synchronization is essential for timing the opening of fuel injectors and the ignition of the spark plugs. The supplied torque and the emitted pollutants depend crucially on the accuracy of these operations.

7.2 Task architecture

In order to evaluate the performance of the resource sharing algorithms for our target application, we need a model view representing the thread architecture. The view must define the typical abstractions used in schedulability

[3]For engine position we mean both the angular position of the crankshaft and the working phase (i.e. intake, compression, expansion or exhaust) of each cylinder.

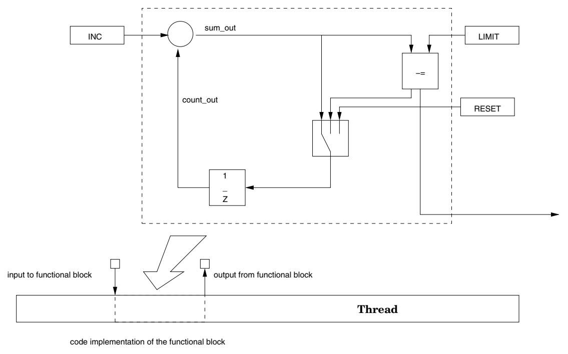
INC
sum_out
LIMIT
−=
count_out
RESET
1
Z
input to functional block
output from functional block
Thread
code implementation of the functional block

Figure 2. A thread contains the implementation of several functional blocks

analysis, such as the real-time threads, each characterized by its activation mode (periodic or sporadic), and its timing characteristics (such as the WCET) and the shared resources used by the threads, with the execution times of the methods called upon them.

An extremely short introduction to a typical automotive software development process can help understanding the nature of the application threads and the relationships between them and the set of shared resources. The threads running under the control of the RTOS are the result of a software development process, which starts from the definition of a high level model (usually a functional model obtained from a tool like Simulink) and continues with the automatic production of software code implementing the functional blocks defined in the model. The code implementing the functional blocks is statically scheduled in the context of a thread (see figure 2). As a result, each thread performs many read and write accesses to the input and output variables or devices defined by the function blocks implemented in it. These sets of input and output variables/devices are possibly implemented as shared resources and the resulting graph of use relationships among tasks and resources is quite densely connected, with each task accessing many resources.

Unfortunately, the exact specification of the application architecture and its performance/timing data (as implemented in the current version of the controller) are considered extremely sensitive industrial property. Furthermore, the current implementation is on a single-cpu controller and it is expected that it will change when ported to the new Janus architecture. Given this restriction, our analysis had to settle for realistic data on the application threads and resources, which could be used for measuring the quality of the algorithms and comparing their performance. The model view we analyzed can be considered a good abstraction of the current implementation and the starting point for evaluating algorithms and solutions (on the worst-case side) for the upcoming Janus implementation.

Based on the analysis of the current implementation and based on the number and complexity of the function points in the new implementation, we considered from 10 to 20 periodic tasks and from 2 to 6 aperiodic tasks with periods ranging from 1 ms to about 100 ms (given the dependency from specification requirements, such as the maximum rpm of the engine, the rate requirements should be considered quite reliable data). Tasks are divided into 3 classes:

high rate: from 1 ms to 5 ms.

medium rate: from 5 to 20 ms.

low rate: from 50 to 100 ms.

Tasks are distributed among the three classes in this way: 50% of the tasks belong to the medium rate type, the other types account for 25% of the tasks each. The processors are quite heavily utilized, utilization ratios above 70% should be expected for each processor. The fraction of the processor utilization required by each class is the following: 50% of the processor time is used to serve high rate tasks, 30% is allocated to the medium rate class, and the last 20% will be used to execute the low rate class. As for resources, tasks share both physical and logical resources. The Janus physical resources shared by tasks are the I/O channels for Analog to Digital (A/D) conversion and the serial ports that are used for communication with the outside systems. The logical resources consist of memory buffers for communication. Both kind of resources are protected by priority ceiling (MSRP or MPCP) semaphores.

Access to the shared I/O channels can be characterized as follows: the serial bus is expected to work at high speeds (the target rate is 500 kb/s) transmitting one byte at a time, corresponding to about 50 μs of required access time. The serial communication will be used only once for each task. Two serial ports (UARTs) are implemented in Janus. The A/D conversion device can be used multiple times, from 5 up to a maximum of 10 times for each task. The A/D access time is dominated by the setup time, resulting in critical sections of about 5 μs.

Tasks are expected to communicate through shared memory resources, which are of two types: switched (no-wait) buffers and one-position mailboxes. Resources of the first type do not actually need semaphores, since the pointer swapping instruction is provided as an atomic instruction by the ARM processors and only one writer task is expected for these resources. As for the second type of resources, tasks are expected to cooperate by exchanging information on their internal state as a set of shared variables. These sets consist of 20 to 50 sets, each one containing between 10 and 300 variables, which must be written and read atomically, in order to keep the state consistent. Each variable is implemented using a 16 or 32 bit data type.

These shared memory requirements actually represent a worst case approximation and in no case will the overall memory allocated to shared variables exceed an architecture-specific bound of 16 KBytes. Write operations are expected to affect all the variables in the data set, and read operations only address a subset (uniformly distributed between 10% and 100%) of the variables in the set. Each task is expected to perform from 3 up to 20 read accesses and from 2 to 8 write accesses to the sets of variables. Finally, in order to ease the schedulability of the task set, a large percentage of the resources accessed by high rate tasks is implemented by using switched buffers (when allowed by the communication semantics).

8 Experimental results

8.1 Generic task sets

In the first set of experiments we compare the performance of the MSRP and MPCP algorithms on a range of task configurations (random load) mapped on the 2 Janus processors.

The experiments consider a set of 6 to 10 tasks statically allocated to each CPU. Depending on the experiments, task periods are chosen randomly between 1 and 100 or by selecting appropriate harmonic values. Harmonic periods are generated in the following way: the period of the first is 1; the period of the next task is given by the period of the previous task multiplied by a random factor between 1 and 4 (ratio 1 has the 30% of probability, ratios 2,3,4 share the remaining 70%).

If U is the system utilization, defined as $U = \Sigma_i c_i/\theta_i$, the total load on each CPU ranges from U_{min} and U_{max}, where U_{min} ranges from 0.025 and 0.925 with steps of 0.025, and U_{max} ranges from Umin+0.025 to 0.95 with steps of 0.025. The number of local resources is always 6 for each processor, plus 6 global resources. The number of critical sections accessed by each task is a random value chosen in the intervals (0,2), (1,4), (2,6) depending on the experiment. Tasks spend a percentage of their computation time into critical sections. The fraction of execution time that is spent in a critical section (local or global) ranges between C_{min} and C_{max}, where C_{min} and C_{max} belong to the set $\{0.0, 0.5, 0.10, 0.15, 0.2\}$, and C_{max} is always greater than C_{min}. For each task set we consider a set of 101 possible configurations, obtained considering that the time spent in a critical section is allocated for a percentage x to local critical sections, and for a percentage (1-x) to global critical sections, with x ranging from 0 to 1 with steps of 0.01. On each configuration we check if the MSRP and the MPCP tests can guarantee the task set as schedulable (more than 520 million configurations were tried).

The first set of experiments is performed on task sets where periods are randomly chosen. The graphs show the percentage of tasks sets that can be guaranteed to be schedulable. It is easy to see how the MSRP policy performs better than MPCP mainly because of the higher utilization bound of EDF when compared to Rate Monotonic.

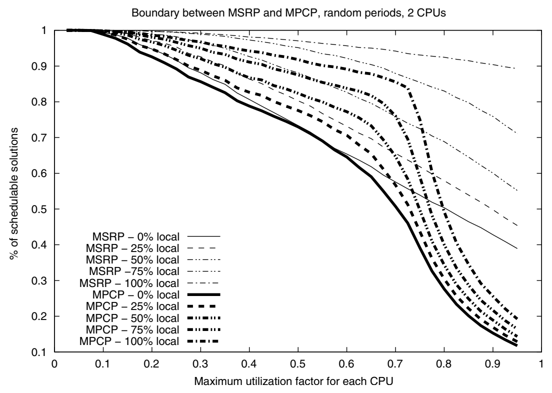

Figure 3. Percentage of schedulable solutions, random periods, variable percentage of local resource utilization.

For higher utilizations the guarantee rate decreases, most notably for the MPCP solution, where it finally approaches a hyperbolic bound for higher values (the hyperbolic bound for rate monotonic scheduling defined in [3] is used).

In general, in all our experiments on random task periods, MSRP always performed better. Even if this is mostly due to the use of EDF as a task scheduling policy, it is our opinion that this advantage should not be easily dismissed. A comparison which does not give an a priori advantage to MSRP because of the higher schedulability bound of EDF can be obtained by selecting task sets where periods are harmonic, therefore having a utilization bound of 1 for the Rate Monotonic policy. The results for this case (Figure 4) show that there is no algorithm performing better on the whole scale of the spectrum (for all the possible percentages of local resources). As one should expect MPCP performs better for a higher percentage of global resources while MSRP is better if a greater percentage of local resources is simulated.

The MPCP curves are always between the minimum and the maximum curves for MSRP. This implies the existence of a crossing point which identifies the percentage of local access to resources that, for each U separates the zone where MPCP performs better from the range where MSRP guarantees an higher percentage of schedulable sets.

To better highlight these regions it is useful to plot the data with a different X axis variable: the percentage of local resource utilization. For example, in Figure 5 MSRP outperforms MPCP for high local resource usage, that is when at least 40% of the resource access time is on local resources (the upper right region in Figure 5).

It can be noted that, as U increases, the lines decrease (since the system load is greater, fewer schedulable solutions can be found). Moreover, when the use of global resources increases (X axis going to 0) there is a point (the boundary in the figure) where MPCP starts to perform bet-

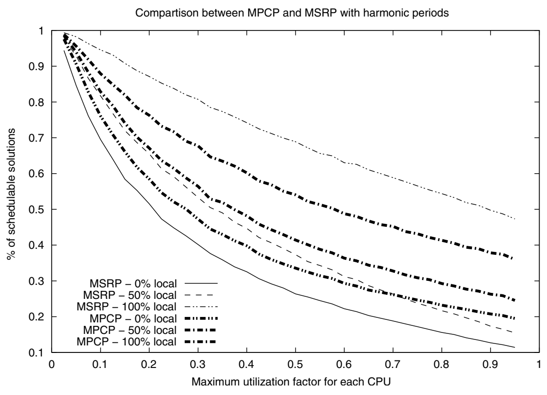

Figure 4. Percentage of schedulable solutions, harmonic periods, variable percentage of local resource utilization.

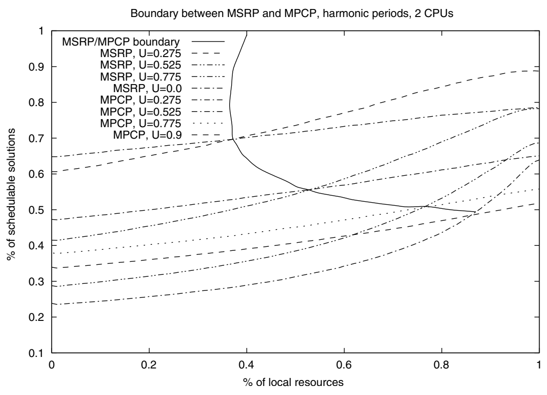

Figure 5. Comparison of MPCP and MSRP with the performance boundary (Y=percentage of schedulable solutions, X=percentage of local critical sections).

ter (which can be explained because the spin locking term influences not only the blocking time, but also the task computation time). A continuous spline, interpolating the crossing points in the figure gives an idea of the boundary between the areas where the two algorithms perform better.

Experiments clearly show how the area where the MSRP protocol performs better increases for a higher use of shared resources. This is a side effect of the reduction of schedulability caused by a higher use of shared resources. In case of Figure 6, the lines are not simply splines, but are the result of comparative experiments for points of the plane (U, % of local resources).

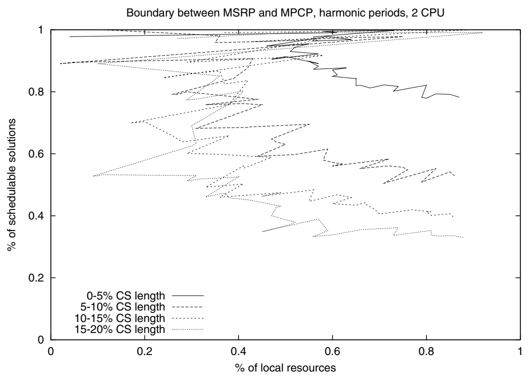

Figure 6. Boundary obtained considering 2 CPUs with various resourse usages.

8.2 The power-train case

The results of our experiments on generic task sets can hardly be considered conclusive for the harmonic periods case. In our second set of experiments we focused on our power-train specifications, to see if more knowledge could be gained when restricting the application domain.

The task sets used for the evaluation of our power-train case study were created using the abstract architecture specification defined in Section 7.

Utilization A set of experiments was performed for different values of the system (2 CPUs) utilization. We considered utilization values from 1.4 to 1.96 with steps of 0.04. In our graphs, the utilization value is the variable on the X axis.

Tasks The total number of tasks in each experiment is a random variable with integer values uniformly distributed in the interval $[12, 26]$. Tasks are divided in three subclasses according to their rate of execution. We generate tasks with random periods and with harmonic periods. Periods have integer values (in msecs). Worst case execution times are chosen in a way that the utilization of each class of tasks sums to the desired value for the class. Task allocation is performed by a simulated annealing algorithm (described in [5]).

Resources and Critical Sections Physical resources are modeled as follows. The Janus chip has two serial ports (UARTs). We assume each serial port is allocated to the tasks running on one of the CPUs. In this way the serial port is a resource shared only among local tasks. Resource index 0 is reserved to the "Serial I/O" channel. The critical sections that use the UART are assumed as 50 μs long. Resource 1 is the "A/D converter". The critical sections that use this resource are 5 μs long. The remaining resources are shared "memory resources". Given the requirements of Section 7, each memory resource uses from 20 to 1200 bytes of memory. For our target (ARM-based) Janus platform, the maximum length of a critical section is estimated as $4 + \frac{83-4}{1200-20} * memorysize$ μs[4]. The simulator loop that generates the resource sets stops at the 16 Kbytes limit and the critical sections computed in step 4 are accepted until the total critical section time is lower than the task WCET. Finally, every critical section accessed by a high rate task has a 40% probability to be deleted, to account for the fact that high priority tasks will use switched buffers when possible in order to reduce their blocking time.

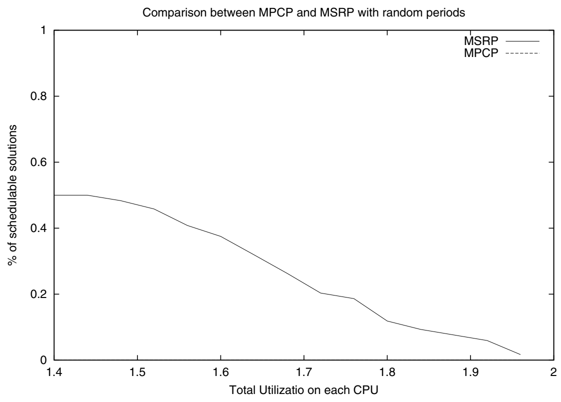

Figure 7. Percentage of schedulable task sets with randomly selected periods on Janus by MPCP/MSRP.

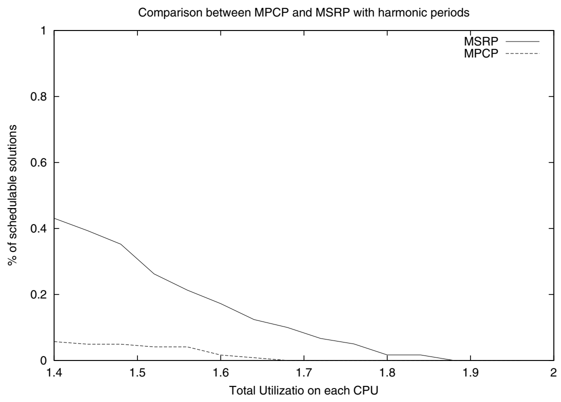

Figure 8. Percentage of schedulable task sets with harmonic periods on Janus by MPCP/MSRP.

[4] 4 to 83 μs is the expected time to write the data on a 40Mhz Janus platform.

8.2.1 Results

We ran experiments for increasing processor utilization factors from 1.4 (approximately 0.7 for each CPU) to 2.0. First, sets of tasks with random integer periods were tried. After processing about 6000 task sets generated according to our specifications, we obtained the results shown in Figure 7. This time the performance difference between the two algorithms is striking: not a single task set is found schedulable with MPCP and the schedulability ratio provided by MSRP goes down (almost linearly) from about 50% at 1.4 utilization (0.7 for each CPU) to about 0 at 1.98 utilization. In the graph of Figure 7, the MPCP curve is not visible since it is completely hidden by the X axis. The situation does not improve significantly for MPCP when the task periods are forced to be harmonic. The MPCP guarantee ratio goes barely up for 1.4 utilization but it is always below 10%. No schedulable set is found under MPCP for utilization values higher than 1.7. In contrast, MSRP continues to deliver an acceptable performance going from more than 40% of schedulable sets at 1.4 utilization, to virtually no schedulable solution at 1.8/1.9 utilization.

When compared with the generic task graphs tried in the previous set of experiments, our power-train case study has at least two striking differences:

- Each critical section is quite short when compared to the execution time of the tasks. In our power-train case high rate tasks spend up to 20% of their time while accessing critical resources, but the time spent by medium and low rate tasks is significantly lower. Furthermore, in our test case, the time spent in each critical section is quite small when compared to our previous experiments, since tasks perform more accesses but with shorter execution times. In the context of the results on the generic task sets this means we expect our power-train application to be in the range of quite low resource usage.

- Each task uses many resources and each resource is accessed by many tasks. In our previous case, tasks used from 0 up to a maximum of 6 critical sections each. In the power-train case there is a much more connected graph of task-resource use relationships. In turn, this means more pessimism in the evaluation of the worst case assumptions of MPCP, since the factors n_i^G, $NC_{i,j}$ and $NH_{i,r,j}$ from which the blocking factors of MPCP depend linearly are now significantly higher. On the other side, the blocking factors of MSRP depend only on the worst case length of individual critical sections.

Since we expect both characteristics to be quite common for automotive applications developed according to the guidelines described in Section 7 we expect MSRP to retain a significant advantage over MPCP even under significant changes in the number of task and/or resources in the final implementation.

9 Conclusions and Future work

Using spin-lock for accessing mutual exclusive resources in real-time multi-processor systems can possibly lead to a non-schedulable system, because the worst case execution time of a task is increased while keeping the processor idle. When we were faced with the problem of designing a concurrency control protocol for the multiprocessor Janus platform, our goal was to obtain a simple and effective algorithm that allows extending the SRP protocol and therefore, permits to share the stack among all the tasks that are allocated to one processor. As a solution, we proposed a spin-lock mechanism for accessing global resources. We expected an advantage in terms of implementation complexity and a disadvantage in terms of schedulability.

After performing an extensive set of simulations, we discovered that the spin-lock mechanism is not necessarily a disadvantage, but performs even better (in terms of schedulability guarantees) for given application contexts. Our simulations show that no algorithm outperforms the other on the whole spectrum of the possible task sets. Which one is the best in terms of the schedulability bound depends on the characteristics of the task set: when access to local resources is clearly dominating with respect to the use of global critical sections, and when the critical sections are short, MSRP presents a better schedulability bound than MPCP.

A second set of experiments, performed on a power-train case study, clearly showed how MSRP can guarantee a higher percentage of task sets when compared to MPCP.

Finally, even in the cases in which MPCP is better than MRSP, it should be considered that MSRP is very simple to implement and has a lower overhead than MPCP. In the Janus case, simplicity and memory optimization were the primary goals.

Regarding other possible approaches to resource sharing, an interesting possibility is to use lock-free algorithms. Lock-free approaches to real-time scheduling were proposed by Anderson, Ramamurthy and Jeffay in [1]. In this approach, a task can execute a critical section more than once, because of the possible conflicts during access. However, when considering periodic real-time tasks, the number of retries is bounded. Intuitively, these approaches can be used especially for short critical sections. However, a deeper study is needed.

References

[1] J. Anderson, S. Ramamurthy, and K. Jeffay, Real-Time Computing with Lock-Free Shared Objects ACM Transactions on Computer Systems, Volume 15, Number 2, pp. 134-165, May 1997.

[2] T.P. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3, 1991.

[3] Enrico Bini and Giorgio Buttazzo and Giuseppe Buttazzo. A Hyperbolic Bound for the Rate Monotonic Algorithm. *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems,2001*

[4] A. Ferrari, S. Garue, M Peri, S. Pezzini, L.Valsecchi, F. Andretta, and W. Nesci. The design and implementation of a dual-core platform for power-train systems. In *Convergence 2000*, Detroit (MI), USA, October 2000.

[5] Paolo Gai and Giuseppe Lipari and Marco Di Natale. Minimizing Memory Utilization of Real-Time Task Sets in Single and Multi-Processor Systems-on-a-chip. *Proceedings of Real-Time Systems Symposium, 2001*

[6] R.L. Graham. *Bounds on the performance of scheduling algorithms*, chapter 5. Coffman Jr. E. G. (ed.) Computer and JobShop Scheduling Theory, Wiley, New Yorj, 1976.

[7] J. Y. T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2:237–250, 1982.

[8] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1), 1973.

[9] Yingfeng Oh and Sang H. Son. Allocating fixed-priority periodic tasks on multiprocessor systems. *Journal on Real Time Systems*, 9, 1995.

[10] R. Rajkumar. Synchronization in multiple processor systems. In *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[11] Manas Saksena and Yun Wang. Scalable real-time system design using preemption thresholds. In *Proceedings of the Real Time Systems Symposium*, December 2000.

[12] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE transaction on computers*, 39(9), September 1990.

[13] K. Tindell, A. Burns, and A. Wellings. Allocating real-time tasks (an np-hard problem made easy). *Real-Time Systems Journal*, 1992.