Journal of Systems Architecture 60 (2014) 258–270

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Exploring the design space of multiprocessor synchronization protocols for real-time systems

Andreu Carminati [a,*], Rômulo Silva de Oliveira [a], Luís Fernando Friedrich [b]

[a] Universidade Federal de Santa Catarina, DAS-CTC-UFSC, Caixa Postal 476, Florianópolis, SC, Brazil
[b] Universidade Federal de Santa Catarina, INE-CTC-UFSC, Caixa Postal 476, Florianópolis, SC, Brazil

ARTICLE INFO

Article history:
Available online 7 December 2013

Keywords:
Real-time
Scheduling
Synchronization
Multiprocessors

ABSTRACT

The goal of this paper is to explore the design space of protocols for multiprocessor systems with static priority and partitioned scheduling. The design space is defined by a set of characteristics that can vary from one protocol to another. This exploration presents new protocols with different characteristics from existing ones. These new protocols are considered variations of the Multiprocessor Priority Ceiling Protocol (MPCP), but they can also be seen as variations of the Flexible Multiprocessor Locking Protocol (FMLP), since they include features common to both protocols. Schedulability tests are provided for these new variations and they are compared with the original versions of MPCP and FMLP. Such comparisons include an empirical comparison of schedulability and an overhead evaluation of a real implementation. Such comparisons show that these new variations are actually competitive in relation to the existing protocols.

1. Introduction

In recent years, the increasing use of multiprocessor systems has motivated the fast development of many solutions for the scheduling of tasks in these systems in the presence of real-time requirements. One of the greatest challenges of real-time scheduling is to synchronize the access to mutual-exclusive resources efficiently, since that the existing synchronization protocols for uniprocessor systems can not be directly extended to multiprocessors. In this context, to schedule a set o real-time tasks on a multiprocessor system, we must limit the blocking time of each of these tasks using an appropriated synchronization protocol. Without a bounded blocking time, a task cannot have its deadline restrictions guaranteed.

The goal of this paper is to explore the design space of synchronization protocols for real-time systems in multiprocessor environment. The design space is considered as a set of basic characteristics, that include: queuing policy in case of blocking (FIFO, priority order, etc.), preemptability of critical sections, and execution control policy in case of blocking (suspension or spin.) In this context, a protocol can be seen as a carefully selected set of characteristics from the design space of protocols. Existing protocols do not explore all possible features within the design space. With this exploration of characteristics as premise, we propose new protocols that will be treated as variations of the Multiprocessor Priority Ceiling Synchronization Protocol (MPCP.) The MPCP was originally proposed by Rajkumar et al. [1,2] for multiprocessor systems with static priority and partitioned scheduling, where tasks are statically allocated to processors. These new variations can also be seen as variations of FMLP [3], since they include features common to both protocols.

Schedulability tests are provided for these new variations and they are compared with two existing protocols for the same system model (partitioned and static priority scheduling): the Multiprocessor Priority Ceiling Protocol for Shared Memory (MPCP) and the Flexible Multiprocessor Locking Protocol (FMLP). In this paper we consider only global resources (which are accessed by more than one processor) in both equations and in the empirical comparison of protocols. Local resources can be handled by protocols designed for uniprocessor systems, which has been extensively studied in the literature. This is the usual approach of the literature on multiprocessor synchronization protocols.

The objective of this paper is not to propose better protocols than the existing ones for all system configurations. Previous studies have shown that, for multiprocessor systems, there is not a synchronization protocol that dominates all others in all situations (different task sets and system models). It is always possible to hand craft a task set that favors one or another existing protocol. We present in this paper variations of the MPCP that improve in some cases the system schedulability reducing the number of processors needed to schedule a system. Some variations are actually simplifications that favor the implementation in real systems.

* Corresponding author. Tel.: +55 04896318322.
E-mail address: andreu@das.ufsc.br (A. Carminati).

1383-7621/$ - see front matter © 2013 Elsevier B.V. All rights reserved.
http://dx.doi.org/10.1016/j.sysarc.2013.11.010

The rest of the paper is organized as follows: Section 2 presents the system model and notations used throughout the text. In Section 3 we present a review of the MPCP protocol. In Section 4 and 5 we do the same for the FMLP and MSRP protocols, respectively. Section 6 presents our proposals. In Section 7 we show a comparison between the proposed solutions and existing protocols. Section 8 presents an implementation of one variation, which will be used in Section 9 for overhead evaluation. Finally, Section 10 presents the conclusions of this paper.

2. System model

In this work it is considered only partitioned and static priority scheduling of periodic task sets. In partitioned scheduling, tasks are statically allocated to processors via dedicated queues. Global scheduling will not be considered in this work. The partitioned scheduling with static priority is also known as P-SP (partitioned static priority). This type of scheduling is addressed by uniprocessor algorithms in each processor, so the main problem is how to perform the partitioning of the tasks. The partitioning of the tasks among processors is effectively the Bin Packing problem, as pointed by Coffman et al. [4], which is NP-hard with combinatorial complexity. One possible strategy is to partition according to some heuristics, such as BFD (Best-fit decreasing), RM-FFDU described by Oh et al. [5] or group by resource usage as in [6]. In this work we do not explicitly consider nested critical sections and deadlock avoidance. It is perfectly possible to do as in [3] where techniques such as group locking are used with synchronization protocols to allow nested critical sections.

2.1. System definition

The notation that will be used throughout the text is the same as in [6], because we based the schedulability analysis of the protocols in that article. Task τ_i is defined by the tuple $((C_{i,1}, C'_{i,1}, C_{i,2}, C'_{i,2}, \ldots, C'_{i,s(i)-1}, C_{i,s(i)}), T_i)$ where:

- $s(i)$ is the number of normal execution segments of τ_i, whereas $s(i)-1$ is the number of critical section segments of τ_i.
- $C_{i,j}$ is the WCET of the jth normal execution segment, whereas $C'_{i,j}$ is the WCET of the jth critical section segment.
- C_i is the total WCET of task τ_i. In this notation, the execution time of each task is segmented into sections of normal execution and critical sections. The worst-case execution time of a task τ_i can be calculated as the sum of all task segments, being they normal or critical. The task execution time can be calculated by the following equation:

$$C_i = \sum_{j=1}^{s(i)} C_{i,j} + \sum_{k=1}^{s(i)-1} C'_{i,k}. \tag{1}$$

- $\tau_{i,j}$ is the jth normal execution segment of a task τ_i, whereas $\tau'_{i,j}$ is the jth critical section segment of a task τ_i.
- T_i is the period of τ_i.
- $R(\tau_{i,j})$ is the resource corresponding to the jth critical section segment of a task τ_i.
- $P(\tau_i)$ is the processor that executes task τ_i.Given two tasks τ_i and τ_j, if $i < j$ then the priority of τ_i is higher than the priority of τ_j.

2.2. Blocking aware response time analysis

Based on the notation presented and the blocking times (which will be presented later), one can calculate the worst-case response time (WCRT) W_i of a task τ_i.

The equations described in this section were presented in [6,7]. The WCRT of a task is calculated iteratively:

$$W_i^{n+1} = C_i + B_i^r + I_i^n + B_i^{low}. \tag{2}$$

In Eq. (2), C_i is the task WCET, B_i^r is the total remote blocking time (that is dependent of the synchronization protocol utilized), I_i^n represents the interference imposed by higher priority tasks. B_i^{low} represents the blocking time imposed by lower priority tasks. This blocking time exists because a task can be prevented from executing at its release as the result of a lower priority task running on a non-preemptive way (inside a critical section in this case). The initial value of the convergence must be $W_i^0 = C_i + B_i^r$.

The calculation of interference follows two approaches, one for suspension-based protocols and another for spin-based protocols:

- For suspension-based protocols (tasks are suspended and they do not spin-lock), the calculation of interference must be done by Eq. (3).

$$I_i^n = \sum_{h<i \& \tau_h \in P(\tau_i)} \left\lceil \frac{W_i^n + B_h^r}{T_h} \right\rceil C_h. \tag{3}$$

In Eq. (3), the blocking times of a higher priority task may increase the WCRT of a lower priority task. This equation captures an effect called back-to-back execution pointed by Rajkumar [8] and Lakshmanan et al. [6], where a task suffers additional interference from auto-suspending (when blocked on some mutex) higher-priority tasks (not accounted in normal interference scheduling analysis). This happens because a higher priority task can suspend itself in the middle of its execution and come to preempt a lower priority task more than once during its activation. An upper bound for this type of interference is B_h^r, that must be added to W_i^n in a way similar to release jitter modeling.

- For spin-based protocols, the calculation of interference must be done by the following equation:

$$I_i^n = \sum_{h<i \& \tau_h \in P(\tau_i)} \left\lceil \frac{W_i^n}{T_h} \right\rceil (C_h + B_h^r). \tag{4}$$

In Eq. (4), the spin time of a task (when blocked) appears as an increasing on its own computing time from the standpoint of lower priority tasks. So, this time must be summed to the computing time of each higher priority tasks, as can be seen in Eq. (4).

The calculation of the blocking time caused by lower priority tasks must follow three approaches, one for suspension-based protocols and two for spin-based protocols:

- For suspension-based protocols, such calculation can be done by the following equation:

$$B_i^{low} = s(i) \times \sum_{l>i \& \tau_l \in P(\tau_i)} \max_{1 \le k < s(l)} C'_{l,k}. \tag{5}$$

In Eq. (5), every time a task blocks (in the worst-case, a task will block on every resource acquisition attempt) and also before its activation, it will allow lower priority tasks to execute. This lower priority tasks can block on resources. When the lower priority task receives the resource, it will preempt the higher priority tasks, because all synchronization protocols that use suspension execute critical sections with a priority higher than normal priorities. This equation offers an upper bound for this blocking.

- For spin-based protocols, this calculation varies whether the protocol treats blocking with preemptive or non preemptive spin: