2009 30th IEEE Real-Time Systems Symposium

Coordinated Task Scheduling, Allocation and Synchronization on Multiprocessors

Karthik Lakshmanan[†] Dionisio de Niz[‡] Ragunathan (Raj) Rajkumar[†]
klakshma@ece.cmu.edu, dionisio@sei.cmu.edu, raj@ece.cmu.edu
[†]Electrical & Computer Engineering and [‡]Software Engineering Institute, Carnegie Mellon University

***Abstract*—Chip-multiprocessors represent a dominant new shift in the field of processor design. Better utilization of such technology in the real-time context requires coordinated approaches to task allocation, scheduling, and synchronization. In this paper, we characterize various scheduling penalties arising from multiprocessor task synchronization, including (i) blocking delays on global critical sections, (ii) back-to-back execution due to jitter from blocking, and (iii) multiple priority inversions due to remote resource sharing. We analyze the impact of these scheduling penalties under different execution control policies (ECPs) which compensate for the scheduling penalties incurred by tasks due to remote blocking. Subsequently, we develop a synchronization-aware task allocation algorithm for explicitly accommodating these global task synchronization penalties. The key idea of our algorithm is to bundle tasks that access a common shared resource and co-locate them, thereby transforming global resource sharing into local sharing. This approach reduces the above-mentioned penalties associated with remote task synchronization. Experimental results indicate that such a coordinated approach to scheduling, allocation, and synchronization yields significant benefits (as much as 50% savings in terms of required number of processing cores). An implementation of this approach is available as a part of our RT-MAP library[1], which uses the `pthreads` implementation of Linux-2.6.22.**

I. Introduction

Hardware limitations impose theoretical restrictions on processor clock speeds. Modern processor technology is already edging towards these physical barriers. Industry consensus appears to be that scaling the number of on-chip processors is the way forward. Most available chip-multiprocessor[2] offerings resemble classical multi-processor architectures, generating renewed research interest in their study. Existing software mechanisms need to be carefully revisited to develop scalable solutions to support future chip-multiprocessors with tens of processing cores. In the context of real-time systems, we specifically require better collaboration among task allocation, scheduling, and synchronization.

Allocating tasks to processors is a well-studied problem in the realm of real-time systems. In general, tasks may be either statically allocated to processors (partitioned scheduling) or dynamically allocated from a single shared task-queue (global scheduling). Global scheduling algorithms can reach higher theoretical utilization bounds than partitioned approaches. However, they require job migration which can incur significant overheads on partitioned memory architectures. On chip-multiprocessor architectures with multiple levels of a private cache for each processing core, the penalty of inter-core task migration is high. Partitioned scheduling algorithms are therefore preferred on such offerings. As we show later, the use of partitioned scheduling also enables us to colocate tasks accessing shared resources, thereby reducing the scheduling penalties associated with global task synchronization.

We focus on partitioned fixed-priority scheduling due to:

1) The simplicity of the scheduler: enabling it to be implemented on devices like interrupt controllers

2) Wide support in commercial real-time operating systems like VxWorks, LynxOS, ThreadX ([1]), and

3) Readily available interfaces in industrial standards like POSIX and AUTOSAR [2]

Some users also perceive the criticality of a task as its scheduling priority, and fixed-priority scheduling naturally accommodates such schemes. For a detailed discussion, see [3].

Task synchronization is an important problem faced in the context of multi-tasking systems. Available solutions in the uniprocessor context like the Priority Ceiling Protocol (PCP) [4] have been extended to the multiprocessor scenario [5, 6]. In this paper, we detail some of the scheduling penalties arising due to multiprocessor task synchronization, and analyze them under different execution control policies. Subsequently, we develop a synchronization-aware partitioned fixed-priority scheduler to accommodate these inefficiencies.

Synchronization-agnostic task allocation algorithms can introduce bottlenecks in the system by unnecessarily distributing tasks sharing global resources across different processors. Synchronization-agnostic scheduling can also lead to performance penalties by unnecessarily preempting tasks holding global resources. Therefore, coordination among task scheduling, allocation and synchronization is vital for maximizing the performance benefits of real-time multiprocessor systems.

A. Contributions

The major contributions of this work are as follows:

1) Characterization of key synchronization penalties including (i) blocking delays on global critical sections, (ii) back-to-back execution from blocking jitter, and (iii) multiple priority inversions due to remote resource sharing between tasks allocated to different processors.

2) Development and evaluation of a synchronization-aware task-allocation scheme to accommodate these task synchronization penalties during the allocation phase.

[1]RT-MAP can be downloaded at http://www.ece.cmu.edu/~klakshma/rtmap.

[2]*Chip-multiprocessors* are also referred to as multi-cores in the industry.

1052-8725/09 $26.00 © 2009 IEEE
DOI 10.1109/RTSS.2009.51

3) Analysis of the impact of different execution control policies (ECPs) on global task synchronization. An ECP is a mechanism to compensate for the scheduling penalties incurred by tasks due to remote blocking.

4) Detailed empirical study of the above-mentioned execution control policies, and measurements from our RT-MAP library implementation based on `pthreads`. Our empirical results indicate that coordinated scheduling, allocation, and synchronization yields significant benefits (as much as 50% savings compared to synchronization-agnostic task allocation).

B. Organization

The rest of this paper is organized as follows: Section II reviews related work. Section III details the challenges associated with multiprocessor synchronization. We also review the multiprocessor priority ceiling protocol and describe the scheduling inefficiencies involved in remote task synchronization. In Section IV, we present our synchronization-aware task allocation algorithm and the different ECPs. Subsequently, we also provide a detailed analysis of these ECPs. A detailed evaluation of the synchronization-aware task allocation algorithm and the performance of different ECPs is provided in Section V. Finally, we summarize and provide future research directions for multiprocessor synchronization in Section VI.

II. Related Work

The work in this paper encompasses the areas of task-allocation, real-time scheduling, and synchronization. In the field of task-allocation, it is well-known that the problem of optimally allocating tasks to processors is analogous to bin-packing [7], and hence proven to be NP-hard in the strong sense. Practical solutions employ packing heuristics with known polynomial-time complexity to achieve near-optimal task partitioning across multiple processors. In our specific context of fixed-priority rate-monotonic scheduling, it has been proven that the First-Fit Decreasing (FFD) and Best-Fit Decreasing (BFD) heuristics have an utilization bound of $(n+1)(2^{\frac{1}{2}}-1)$, where n is the number of processors [8]. However, this assumes optimal scheduling on each processor and no dependencies due to resource sharing.

Traditional real-time multiprocessor scheduling algorithms have dealt mostly with independent tasks with no interactions. de Niz and Rajkumar have previously developed a set of partitioning bin-packing algorithms to deploy groups of communicating tasks onto a network of processors [9]. The objective of these algorithms is to minimize the number of processors needed while trying to reduce the bandwidth required to satisfy the communication between these tasks. In this paper, we are concerned with tasks that need to synchronize on (potentially) globally shared resources.

Task synchronization in real-time systems is a well-known problem. Fixed-priority scheduling schemes employ techniques like priority inheritance and priority ceiling protocols to enable resource sharing across real-time tasks. Dynamic-priority scheduling schemes also use mechanisms like the Stack-based Resource Policy (SRP) [10] to handle real-time task synchronization. In the context of fixed-priority multiprocessor scheduling, the priority ceiling protocol has been extended to realize the multiprocessor priority ceiling protocol (MPCP) [11]. Synchronization schemes have also been developed for other related scheduling paradigms like PFair [12]. Multiprocessor extensions to SRP have also been considered and performance comparisons have been done with MPCP [13]. This paper adopts a more holistic approach to partitioned task scheduling by explicitly considering MPCP synchronization penalties during task allocation and investigating the impact of different ECPs.

Recent studies [14, 15] have investigated the performance differences between spin-based and suspension-based synchronization protocols. In these studies, their authors found that spin-based protocols impose a smaller scheduling penalty than suspension-based ones, even under zero preemption costs. While these studies present interesting results, the analyses they used on suspension-based protocols can be substantially improved. In this paper, we have developed new schedulability algorithms for these protocols that are less pessimistic and includes our improvements based on [16]. With this new analysis, we found that the suspension-based protocols in fact behave better than spin under low preemption costs (less than 160μs per preemption) and longer critical sections (15 μs) than those studied in [15].

A. Assumptions

We assume fixed-priority scheduling with tasks having conventional rate-monotonic scheduling priorities. We assume a uniform chip-multiprocessor architecture, where all the processing cores have equal processing capabilities. The worst-case execution-time (WCET) of each task segment and corresponding task periods are assumed to be known ahead of time. We also assume implicit-deadline tasksets, where the deadline of each task equals its corresponding period.

All the critical sections are assumed to be non-nested[3].

B. Notation

Each task is considered to be an alternating sequence of *normal* execution segments and *critical section* execution segments. In the rest of this paper, we use the following notation to describe such tasks: Task i is denoted as τ_i, given by $\tau_i : ((C_{i,1}, C'_{i,1}, C_{i,2}, C'_{i,2}, ..., C'_{i,s_i-1}, C_{i,s_i}), T_i)$.

where,

s_i is # of normal execution segments of τ_i.

$s_i - 1$ is # of critical section execution segments of τ_i.

$C_{i,j}$: WCET of the j^{th} normal execution of τ_i.

$C'_{i,k}$: WCET of the k^{th} critical section of τ_i.

T_i denotes the period of task τ_i (and its implicit deadline).

$\tau_{i,j}$ denotes the j^{th} normal execution segment of task τ_i

$\tau'_{i,k}$ denotes the k^{th} critical section of task τ_i

Tasks are ordered in strictly decreasing order of priorities i.e. $i < j$ implies that task τ_i has higher priority than task τ_j.

[3]Nested critical sections may be accommodated for example using aggregate locks on the outermost critical section [5].

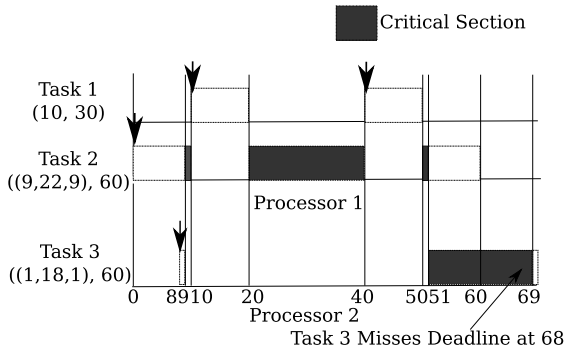

Fig. 1. Penalty of global synchronization

We also use P_l to denote the l^{th} processor. Also, the function $P(\tau_i)$ returns the processor assigned to task τ_i. Similarly, $R(\tau_i, j)$ is used to retrieve the locked resource corresponding to the j^{th} critical section segment of task τ_i.

We will also use the term C_i as a short-hand notation to denote the sum of all the WCET's of task τ_i (both normal and critical section segments). That is,

$$C_i = \sum_{j=1}^{s_i} C_{i,j} + \sum_{k=1}^{s_i-1} C'_{i,k}$$

III. Multiprocessor Synchronization Challenges

In this section, we detail the various challenges associated with synchronization in multiprocessors. We first highlight the key differences between global and local task synchronization.

A. Global vs Local Synchronization

The Priority Ceiling Protocol (PCP) [4] is a real-time synchronization protocol that minimizes the time a high-priority task waits for a low priority one to release the lock on a shared resource, known as blocking time. When PCP is used by tasks deployed on different processors, this blocking time can lead to idling of the processors. For instance, consider Figure 1. In this figure, there are three tasks, τ_1 and τ_2 running in processor P_1 and τ_3 running in processor P_2. In addition, a resource is shared between tasks τ_2 and τ_3 using PCP. The figure depicts how τ_2 locks the resource at time 9 making τ_3 wait for the lock up to time 51 when the lock is released by τ_2. This waiting leaves processor P_2 idle because the only task deployed there, τ_3, is waiting for the lock (known as remote blocking). Furthermore, during the time τ_2 holds the lock it suffers multiple preemptions from the higher priority task τ_1. As a consequence, τ_3 misses its deadline at time 68. Such a problem is removed if, instead of sharing the resource across processors, it is shared on the same processor, i.e., tasks τ_2 and τ_3 are deployed together, say in processor P_2.

The key aspect of the example in Figure 1 is that processor utilization is wasted during remote blocking. This is because a task that could be scheduled in the remote processor is blocked leaving the cycles reserved for it idle. This contrasts with local blocking because the task holding the lock uses the cycles the blocked task leaves idle. Furthermore, such a waste of reserved cycles can be repeated for each blocked task running on a different processor. That is, sharing a resource across n processors can waste reserved cycles in $n-1$ processors, effectively transforming this n processors into a single processor during the execution of the critical section (only one critical section can execute at a time). This highlights the significance of task allocation in determining the schedulability of a task set in a multiprocessor. In other words, the co-location of tasks that lock shared resources to the same processor prevents reserving processors cycles that are wasted in remote blocking. This motivates our synchronization-aware task-allocation algorithm.

In the worst case, however, some degree of global resource sharing may be unavoidable. As a result, techniques to mitigate its consequences are also needed.

B. Multiprocessor Priority Ceiling Protocol

We shall analyze and characterize the behavior of different execution control policies (ECPs) in Section IV. For the sake of self-containment, we present a brief tutorial of the multiprocessor priority ceiling protocol (MPCP) and its properties. Let us start by reviewing some definitions. A *global mutex* is a mutex shared by tasks deployed on different processing cores. The corresponding critical sections are referred to as *global critical sections* (gcs). Conversely, a *local mutex* is only shared between tasks on the same processing core, and the corresponding critical sections are *local critical sections*.

Let J' be the highest priority job that can lock a global mutex M_G. Under MPCP, when any job J acquires M_G, it will execute the gcs corresponding to M_G at a priority of $\pi_G + \pi'$, where π_G is a base priority level greater than that of any other normally executing task in the system, and π' is the priority of J'. This priority ceiling is referred to as the `remote priority ceiling` of a gcs.

MPCP was specifically developed for minimizing remote blocking and priority inversions when global resources are shared. We reproduce below a basic definition of MPCP. Readers should refer to [11] for a more detailed discussion.

MPCP Definition

1) Jobs use assigned priorities unless within critical sections.
2) The uniprocessor priority ceiling protocol [4] is used for all requests to local mutexes.
3) A job J within a global critical section (gcs) guarded by a global mutex M_G has the priority of its gcs ($\pi_G + \pi'$).
4) A job J within a gcs can preempt another job J^* within a gcs if the priority of J's gcs is greater than that of J^*'s gcs.
5) When a job J requests a global mutex M_G. M_G can be granted to J by means of an atomic transaction on shared memory, if M_G is not held by another job.
6) If a request for a global mutex M_G cannot be granted, the job J is added to a prioritized queue on M_G before being preempted. The priority used as the key for queue insertion is the *normal* priority assigned to J.
7) When a job J attempts to release a global mutex M_G, the highest priority job J_H waiting for M_G is signaled and

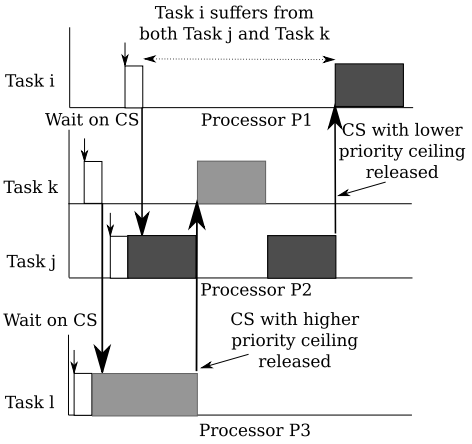

Fig. 2. Transitive Interference during remote blocking

becomes eligible for execution at J_H's host processor at its gcs priority. If no jobs are suspended on M_G, it is released.

Now, consider tasks that must suspend themselves when waiting for global critical section resources. Many penalties are encountered and are described next.

C. Blocking Delay on Remote Resources

MPCP executes all the gcs's at a priority level above all normal execution segments and local critical sections. Even under the purview of such a priority ceiling protocol, it is not possible to guarantee that tasks do not receive transitive interference during remote blocking as shown in Fig. 2. In this figure, Task τ_i in P_1 could be suspended on τ_j in P_2. Task τ_k on P_2 could be suspended on another task τ_l on another processor P_3. The priority ceiling of the mutex shared between τ_k and τ_l could be higher than the priority-ceiling of the mutex shared between τ_i and τ_j. In this scenario, the release of the critical section by τ_l would give control to the task τ_k, which preempts the task τ_j executing its critical section. The task τ_i waiting on τ_j therefore suffers from the interference due to the release of a mutex by τ_l.

1) Back-To-Back Execution of Suspending Tasks: A phenomenon that arises when tasks suspend themselves is that of "back-to-back execution" [17]. Consider the example shown in Fig. 3. There are three tasks $\tau_1 : ((2,2,0),8), \tau_2 : (4,8), \tau_3 : ((1,2,2),64)$. Task τ_1 and τ_2 are assigned to processor P_1. Task τ_3 is assigned to processor P_2.

It is easy to verify that τ_1 and τ_3 are schedulable. However, an anomalous scheduling behavior happens with respect to task τ_2. It should be schedulable if τ_1 follows a periodic release behavior, since it expects at most one preemption from a task with its same period of 8. When τ_2 is released at time instant 3, however, it faces back-to-back execution due to the remote synchronization effect of τ_1 and this leads to τ_2 suffering the interference of a second arrival of τ_1 leading to a deadline miss. This back-to-back preemption arises due to the jitter in the blocking time of task τ_1 and its self-suspending behavior.

2) Multiple Priority Inversions due to Suspension: The key scheduling inefficiency resulting from the remote blocking

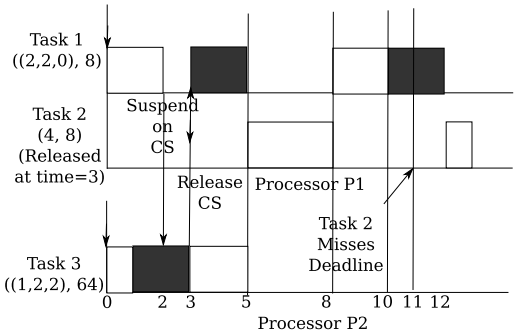

Fig. 3. Back-To-Back Execution due to Remote Synchronization

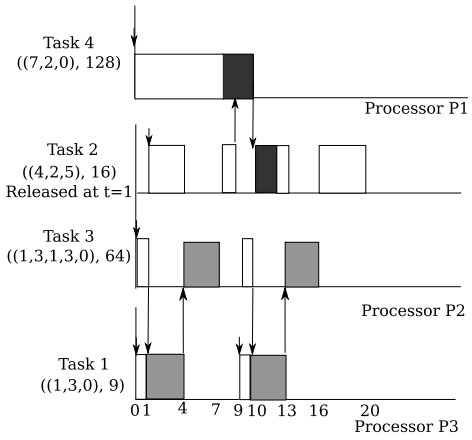

Fig. 4. Multiple Priority Inversions Due to Suspension

behavior of tasks is that of multiple priority inversions due to lower-priority critical sections. For example, consider the scenario shown in Fig. 4. Whenever task τ_2 suspends, task τ_3 can get a chance to execute, and it can request a lock on the global critical section shared with τ_1. When τ_1 releases the global critical section, τ_3 preempts τ_2 due to its higher priority ceiling and interferes with the normal execution of τ_2 *twice*. In the worst case, *every* normal execution-segment (of duration $C_{i,k}$ $1 \leq k \leq s_i$) of a task τ_i can be preempted at most once by each of the lower-priority tasks τ_j $(j > i)$ executing their global critical sections released from remote processors.

IV. Coordination Among Scheduling, Allocation, and Synchronization

Our goal is to develop a holistic scheme which combines a synchronization-aware task allocation strategy, with an efficient protocol for global task synchronization. In order to realize this, we first describe our synchronization-aware task allocation strategy. We then analyze different execution control policies under the multi-processor priority ceiling protocol.

A. Synchronization-Aware Task Allocation

We consider two bin-packing algorithms: the *synchronization-agnostic* and the *synchronization-aware* algorithms. The former packs objects exclusively based on size and the latter tries to pack together tasks that share

mutexes. Both algorithms are modifications of the best-fit decreasing (BFD) bin-packing algorithm. The BFD algorithm orders the bins by non-increasing order of available space and the objects by non-increasing order of object size, and tries to allocate the object at the head of this sorted list into each of the bins in order.

When bin-packing algorithms are used to pack periodic tasks into processors, the utilization of each task is used as its size and one minus the total utilization deployed on a processor is used as the available space in the processors. This approach assumes that the load of the processors can reach 100%, which for rate-monotonic scheduling is only sometimes true. However, in the absence of additional information, such an approach is a good indicator of which task and which processor to try next. In our bin-packing algorithms, once we select the task to be allocated and the candidate processor for trying the allocation, we use our response-time tests to check if this allocation is possible. The fact that the allocation may not be feasible, even if the total utilization is less than 100%, signals the mismatch in assumptions between the packer algorithm and the admission test. Thus, additional inefficiencies are introduced in the packing phase.

When synchronization is used, an additional penalty can be incurred if we distribute the tasks that share a mutex among two or more processors. This is because, if we allocate these tasks to the same processor, the shared mutex becomes a local mutex and local PCP can be used. As described in the previous section, local synchronization eliminates the scheduling penalties associated with global task synchronization.

The strategy of the synchronization-aware packer is two-fold. First, tasks that share a mutex are *bundled* together. This bundling is transitive, i.e., if a task A shares a mutex with task B, and B shares a mutex with C, all three of them are bundled together. Then, each task bundle is attempted to be allocated together as a single task into a processor. We start with just enough processors to allocate the total utilization of all the tasks. Secondly, the task bundles that do not fit are put aside until all bundles and tasks that fit are allocated without adding processors. Now, only bundles that did not fit into any existing processor remain unallocated. The penalty of transforming a local mutex into a global mutex is the additional processor utilization required for schedulability. The cost of breaking a bundle is defined as the maximum of such penalties over all its mutexes. The bundles are then ordered in increasing order of cost, and the bundle with the smallest cost is selected to be broken. This bundle is broken such that it contains at least one piece as close as possible to the size of the largest available gap among the processors (in accordance with the BFD heuristic). If this allocation is not possible, a new processor is added and we try again to partition the task-set. Since the addition of new processors opens up new possibilities to allocate full bundles together, we repeat the whole strategy again starting by retrying to fit the unallocated bundles. In the absolute worst-case, each task may require its own processor, therefore, at most n processors exist in the final packing of any schedulable task-set (where n is the number of tasks).

B. Execution Control Policies

An execution control policy (ECP) is defined as a mechanism to compensate for the scheduling penalties incurred by tasks due to remote blocking. In this work, we consider the following execution control policies:

1) *Suspend:* The task is suspended during remote blocking, enabling lower priority tasks to execute.
2) *Spin:* The task continues to spin on the remote critical section, preventing lower priority tasks from executing.

Other execution control policies such as period enforcement [17] can also be applied for minimizing the scheduling penalty arising from synchronization, but these extensions are beyond the scope of this paper. We now describe different execution control policies and their schedulability implications.

1) MPCP:Suspend: The *MPCP:Suspend* execution control policy forces a task to suspend when it waits for a gcs entry request to be satisfied. In this version, tasks blocking on remote resources release the processor for other tasks executing in the system. It suffers from all the scheduling penalties described in the previous section. We now quantify each of the scheduling inefficiencies described in the earlier sections.

1) Remote Blocking due to Global Critical Sections: This is captured by a separate term $B^r_{i,j}$ for the j^{th} critical section acquired by the task τ_i (the r in $B^r_{i,j}$ denotes *remote* blocking as opposed to *local* blocking). We will compute this term later.

2) Back-To-Back Execution due to Suspending Tasks: In addition to the preemptions considered by conventional Rate-Monotonic Scheduling, in the worst case, back-to-back execution can result in additional interference from each higher priority task (τ_h).

This additional interference is upper bounded by the maximum interference from each higher priority task (τ_h) over a duration equal to its maximum duration of remote blocking ($B^r_h = \sum_{q=1}^{s_h-1} B^r_{h,q}$) (see [16] for details on using such suspension-based blocking terms in response-time tests).

This upper bound on the interference is captured in our analysis (see eqn.(1)) using the second term.

3) Multiple Priority Inversions due to Global Critical Sections: The global critical sections of each lower priority task can affect the normal execution segment of a higher priority task. This is captured by the per-segment interference given by $\sum_{l>i \,\&\, \tau_l \in P(\tau_i)} \max_{1 \le k < s_l} C'_{l,k}$.

Let us denote the total response-time of task τ_i without interference on its local processor as $C^*_i = C_i + B^r_i$. In such a scenario without interference, task τ_i is never preempted on the local processor requiring C_i units of computation time, and it can be remote blocked for at most B^r_i units of time during lock acquisition for global critical sections, resulting in a total response-time of $C^*_i = C_i + B^r_i$.

The worst-case response time of task τ_i is bound by the convergence W_i of:

$$W_i^{n+1} = C_i^* + \sum_{h<i \,\&\, \tau_h \in P(\tau_i)} \lceil \frac{W_i^n + B_h^r}{T_h} \rceil C_h$$

$$+ s_i \sum_{l>i \,\&\, \tau_l \in P(\tau_i)} \max_{1 \le k < s_l} C'_{l,k} \quad (1)$$

where, $W_i^0 = C_i^*$

Let the global priority ceiling of the critical section $C'_{i,k}$ be denoted by $gc_{i,k}$.

The worst-case response time of the execution time of critical segment $C'_{i,k}$ is bounded by:

$$W'_{i,k} = C'_{i,k} + \sum_{\tau_u \in P(\tau_i)} \max_{1 \le v \le s_u \,\&\, gc_{u,v} > gc_{i,k}} C'_{u,v} \quad (2)$$

The reasoning here is that the global critical sections have an absolute priority greater than all the normal execution segments. Therefore, we only need to consider one outstanding global critical section request per task.

$B_{i,j}^r$ represents the remote blocking term for task τ_i in acquiring the global critical section $C'_{i,j}$. This is bounded by the convergence,

$$B_{i,j}^{r,n+1} = \max_{l>i \,\&\, (\tau'_{l,u}) \in R(\tau_i,j)} (W'_{l,u}) \quad (3)$$

$$+ \sum_{h<i \,\&\, (\tau'_{h,v}) \in R(\tau_i,j)} (\lceil \frac{B_{i,j}^{r,n}}{T_h} \rceil + 1)(W'_{h,v})$$

where, $B_{i,j}^{r,0} = \max_{l>i \,\&\, (\tau'_{l,u}) \in R(\tau_i,j)} (W'_{l,u})$

The key benefit of *MPCP:Suspend* is that the idle time during suspension is available for execution by other tasks. The disadvantages however are the penalties of back-to-back executions and multiple priority inversions per task.

An alternative approach is to prevent back-to-back execution and multiple priority inversions by not relinquishing the processor and spinning until the critical section is obtained [15], similar to *MPCP:Spin* defined next.

2) MPCP:Spin: In the *MPCP:Spin* protocol, tasks spin (i.e. execute a tight loop) while waiting for a gcs to be released. This avoids any future interference from global critical section requests from lower-priority tasks, which may be otherwise issued during task suspension. In practice, this could be implemented as *virtual* spinning, where other tasks are allowed to execute unless they try to access global critical sections. In that case, they would be suspended. As a result, the number of priority inversions per task is restricted to one per lower priority task. The back-to-back execution phenomenon is also avoided since the tasks do not suspend in the middle. The time spent waiting for the lock becomes part of the task execution time, therefore, the task never voluntarily suspends during its execution. This improves average-case performance but cannot guarantee worst-case improvements.

The analysis of the *MPCP:Spin* protocol is quite straightforward. An upper bound on the computation time of task τ_i is given by: $C_i^* = C_i + B_i^r$

The worst-case computation time of task τ_i is bounded by the convergence W_i of:

$$W_i^{n+1} = C_i^* + \sum_{h<i \,\&\, \tau_h \in P(\tau_i)} \lceil \frac{W_i^n}{T_h} \rceil C_h^*$$

$$+ \sum_{l>i \,\&\, \tau_l \in P(\tau_i)} \max_{1 \le k < s_l} C'_{l,k} \quad (4)$$

where, $W_i^0 = C_i^*$.

The blocking terms $B_{i,j}^r$ are the same as those given in Equation (3).

As can be seen, spinning reduces the preemptions from global critical sections of lower priority tasks since $\sum_{l>i \,\&\, \tau_l \in P(\tau_i)} \max_{1 \le k < s_l} C'_{l,k}$ is used in Equation (4) instead of $s_i \sum_{l>i \,\&\, \tau_l \in P(\tau_i)} \max_{1 \le k < s_l} C'_{l,k}$ used in Equation (1) for suspension. However, spinning results in additional preemption from higher priority tasks as captured by using the C_h^* term in Equation (4) instead of the C_h term used in Equation (1) for suspension. As a part of our empirical evaluation, we compare these two different execution control policies, and study the realm of applicability of each policy.

C. RT-MAP Library Implementation

We have implemented *MPCP:Suspend* and *MPCP:Spin* as a part of our RT-MAP (Real-Time Multi-core Application Package) Library , which is built around `pthreads` [18] for Linux-2.6.22.

Task allocation is performed using the `pthread_setaffinity_np()` API. The CPU-affinity masks are modified to reflect the allocation of threads CPUs. We use the built-in real-time thread scheduling support in Linux, using the `pthread_setschedparam()` interface. Our implementation provides wrappers to create real-time threads, perform admission control on created threads, and allocate tasks using our synchronization-aware scheme.

The synchronization schemes are implemented as special data-structures with priority queues, which are internally protected using the `pthread_mutex_lock()` and `pthread_mutex_unlock()` interfaces. The priority of the task is raised to the highest priority level before accessing the synchronization data-structures, in order to mask preemptions during updates to the priority queue.

V. Evaluation

In this section, we present an experimental evaluation of the synchronization schemes and their integration with our synchronization-aware bin-packing algorithm. The metric used to compare the effectiveness of each algorithm is the number of bins needed to allocate a given taskset. The fewer the number of bins needed, the better is the performance. In order to study the performance of synchronization algorithms in isolation, we use the synchronization-agnostic packing algorithm. The benefits of using a synchronization-aware packing algorithm for each of these schemes is quantified later.

A. Experimental Setup

All our experiments evaluate how many processors of equal capacity (100% utilization) an algorithm uses to schedule a task set. We compare the number of processors needed among all the algorithms against the optimal packing algorithm. Given that optimal bin-packing is an intractable problem, we start with a fully-packed configuration instead (from a bin-packing standpoint disregarding scheduling inefficiencies). We do this processor by processor by dividing the 100% utilization into a defined number of tasks that would fit this processor perfectly. Each of these tasks is assigned a random utilization that all add up to 100%. Then, their periods are chosen randomly between 10ms and 100ms. Next, their execution time is calculated to match their utilization. Now, given a selected number of critical sections per task, the execution is divided into two types of segments: normal execution and critical section. These segments are arranged starting with a segment of normal execution followed by one of critical section and then another of normal execution. This arrangement continues until the task has the required number of critical sections. Each critical section is associated with a mutex that is locked by some chosen number of other tasks.

B. Comparison of Synchronization Schemes

We explore three main factors that affect the different ECPs: (i) the size of the critical sections, (ii) the number of tasks per processor, and (iii) the number of lockers per mutex.

In order to obtain a conceptual comparison of the different synchronization schemes, we consider their behavior under zero overheads. Overheads can change, and likely decline over time, and are platform-dependent. Later on, we explicitly specify and evaluate the impact of different preemption costs on these different schemes.

The most important factor that affects the different ECPs is the size of a critical section. Figure 5 depicts the results of our experiments with increasing critical section sizes. We create task sets of 8 fully-packed processors with 5 tasks per processor, 2 critical sections per task, and 2 lockers per mutex. Then, we pick critical section lengths from 5μs to 1280μs.

Initially, both *MPCP:Spin* and *MPCP:Suspend* exhibit similar performance. At longer critical section lengths however, *MPCP:Spin* requires many more processors compared to *MPCP:Suspend*. This is mainly due to the processor time lost during spinning on a mutex. In the case of *MPCP:Suspend*, this time is effectively used by the other tasks executing on the same processor. As a general trend, however, both *MPCP:Spin* and *MPCP:Suspend* require more processors with longer critical section lengths. This is mainly due to increasing remote blocking terms with increasing global critical section lengths. In the case of *MPCP:Suspend*, this overhead manifests as longer priority inversions from lower-priority global critical section executions. *MPCP:Spin* experiences a similar overhead due to the increased usage of CPU time during spinning.

Next, we increased the number of tasks per processor, since this has the potential of increasing the number of preemptions for a task and also the number of priority inversions.

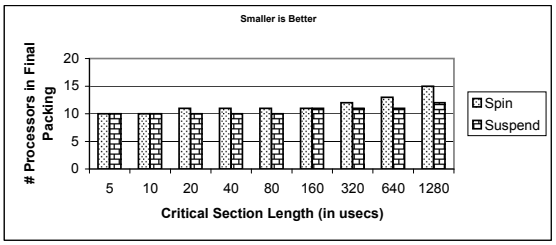

Fig. 5. Efficiency of Algorithms as Length of Critical Section Increases (workload: 8 fully-packed processors, 2 critical sections per task, 2 lockers per mutex, average of 5 tasks per processor in initial workload)

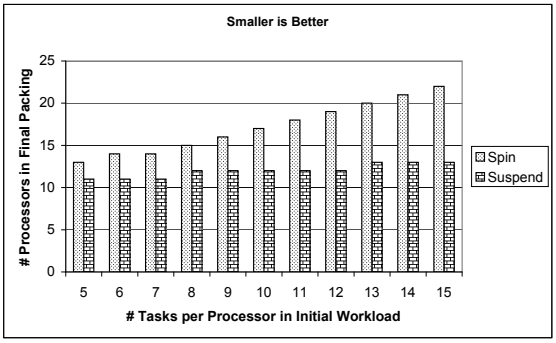

Fig. 6. Efficiency of Algorithms as Tasks per Processor Increases (workload: 8 fully-packed processors, 2 critical sections per task, 2 lockers per mutex, critical section length 500μs)

Figure 6 depicts the number of processors needed to pack a workload of eight fully-packed processors with two critical sections per task and two lockers per mutex. Each critical section has a duration of 500μs. We chose this value to observe meaningful differences between the two ECPs, based on the results of the earlier experiments with varying critical section lengths. Critical sections of similar durations have been observed in the past [19, 20], and hence considering such lengths is both relevant and useful. For smaller critical section lengths, the previous experiments already indicate that the performance difference is negligible, and this was verified.

In Figure 6, we can observe that, as the number of tasks per processor increases, the spin-based synchronization scheme requires more processors compared to the suspension-based scheme. This is because having more tasks during spinning to wait for global mutexes increases the loss of processor utilization. The suspension scheme, however, effectively utilizes this duration to execute other eligible tasks hosted on the same processor. In general, both the schemes require more processors with an increasing number of tasks. With *MPCP:Spin*, this is due to more tasks using CPU time for spinning, whereas with *MPCP:Suspend*, this is due to more priority inversions from lower priority tasks locking global mutexes.

The third factor that affects different ECPs is the number of lockers per mutex. Figure 7 depicts the results of our experiments with a growing number of lockers. For this experiment, we created task sets of eight fully-packed processors with

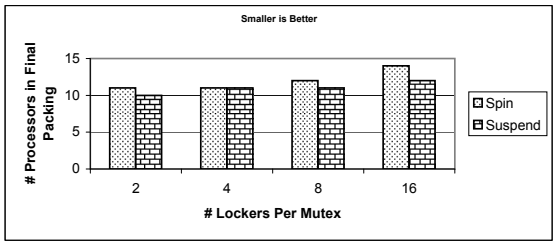

Fig. 7. Efficiency of Algorithms as Lockers per Mutex Increases (workload: 8 fully-packed processors, 2 critical sections per task, critical section length 100μs, average of 5 tasks per processor in initial workload)

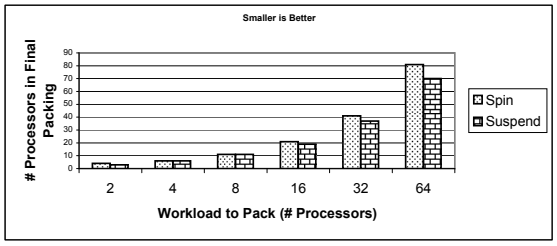

Fig. 8. Efficiency of Algorithms as Workload Increases (2 critical sections per task, 2 lockers per mutex, critical section length 500μs, average of 5 tasks per processor in initial workload)

five tasks per processor, two critical sections per task, and a critical section of 100μs. In this figure, we can see that *MPCP:Spin* is adversely affected, when the number of lockers increases. This is due to the fact that the waiting time can be enlarged with the number of tasks that can potentially lock the mutex. Given that this spinning task does not relinquish the processor, it reduces the total useful utilization. Because *MPCP:Suspend* does not spin-wait for the mutexes, it does not suffer as much. The increase in blocking terms also results in a slight increase for suspension-based schemes. Using less pessimistic admission control tests could potentially improve this scenario for suspension.

Finally, we also explored the trend of increasing the presented workload of fully-packed processors. Figure 8 depicts this experiment. In this figure, we can see that *MPCP:Spin* and *MPCP:Suspend* exhibit similar behavior with increasing workloads. The major observations are: (1) *MPCP:Suspend* requires fewer processors than *MPCP:Spin*, and (ii) the ratio of the number of processors required by the two schemes remains fairly constant. This behavior is due to the fact that increasing the workload only affects the task allocation algorithm as opposed to the synchronization schemes themselves (as long as other parameters remain constant).

In summary, under zero implementation overheads; suspension-based MPCP performs better than spin-based MPCP. However, as we show later in Subsection V-D, implementation overheads also play a significant role in the performance of synchronization protocols. For smaller critical section durations and larger implementation overheads, spin-based MPCP can be expected to perform better because it should have fewer context-switches.

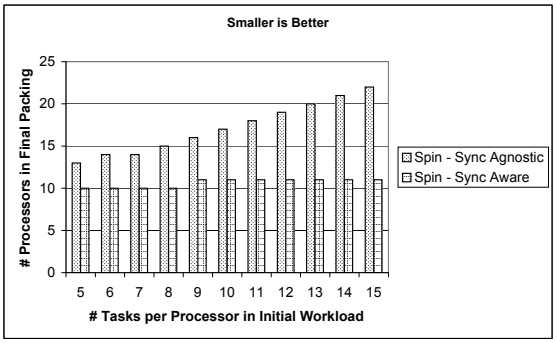

Fig. 9. Efficiency of Sync-Aware Packing with Number of Tasks Per Processor in Initial Workload (Spin) (workload: 8 fully-packed processors, 2 critical sections per task, 2 lockers per mutex, critical section length 500μs)

C. Synchronization-Aware Task Allocation

Our synchronization-aware packing algorithm bundles together synchronizing tasks and tries to deploy them. This strategy reduces the number of global mutexes and resulting remote blocking. However, the bundling heuristic artificially creates larger objects to pack, and this can lead to less efficient packing than the BFD heuristic on the original taskset. As a result, our algorithm does not always pay off. However, when remote blocking penalties play a major role, then our synchronization-aware packer yields significant benefits.

Figures 9 and 10 depict the behavior with increasing number of tasks per processor. In these figures, we pack a workload of eight fully-packed processors, critical section length of 500μs, 2 critical sections per task, and 2 lockers per critical section. As seen in earlier experiments, both schemes exhibited similar performance characteristics at lower critical section lengths, and hence a critical section length of 500μs is considered. On the average, synchronization-aware bin packing saves about 6 processors with *MPCP:Spin*, whereas there is a reduction of about 2 processors with the suspension-based technique.The reason for the bigger savings under spinning is that there is more penalty associated with a global critical section than in a suspension-based scheme (for the critical section length under consideration). Hence, synchronization-aware packing to eliminate such penalties saves more processors in spin-based schemes. As the number of tasks increases, synchronization-aware packing results in up to 50% reduction in the required number of bins (under *MPCP:Spin* beyond 15 tasks per processor). We also investigated the impact of increasing the workload. We again observed much less savings with the suspension-based technique compared to spinning. This is due to the fact that there is more room for optimization under *MPCP:Spin*, compared to *MPCP:Suspend*. The flexibility gained by increasing the workload therefore tends to have an impact on *MPCP:Spin*, whereas there is less benefit for *MPCP:Suspend*.

D. Implementation Cost

We implemented RT-MAP, a `pthreads`-based library for real-time support in multi-core processors. Task allocation

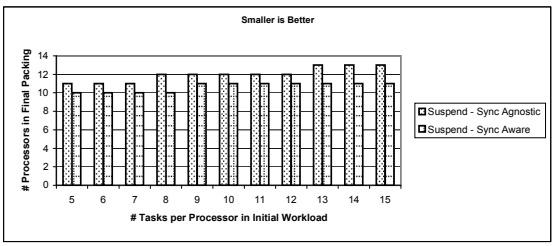

Fig. 10. Efficiency of Sync-Aware Packing with Number of Tasks Per Processor in Initial Workload (Suspend) (workload: 8 fully-packed processors, 2 critical sections per task, 2 lockers per mutex, critical section length 500μs)

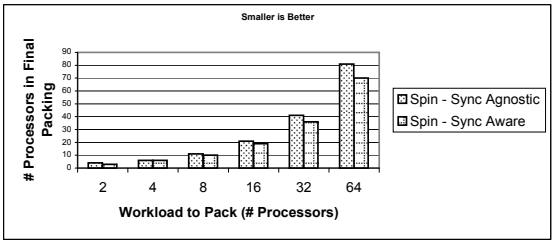

Fig. 11. Efficiency of Sync-Aware Packing as Workload Increases (Spin) (2 critical sections per task, 2 lockers per mutex, critical section length 500μs, average of 5 tasks per processor in initial workload)

and synchronization support are provided through easy-to-use APIs. The implementation costs of the underlying primitives are listed in Table 1. The measurements were performed on an Intel Core 2 Duo processor at 2GHz.

The task allocation algorithm is only performed during initialization, and it therefore does not affect application performance. The values listed in Table 1 only reflect the cost of updating the underlying scheduler data-structures. The major penalty arises from the cost of preemption. However, the true cost of preemption is determined by the effect of preemption on cache content, which is application-specific and depends on the working set-size of application threads.

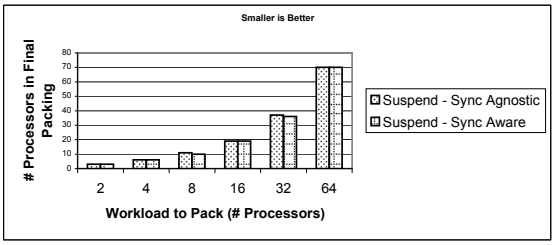

Fig. 12. Efficiency of Sync-Aware Packing as Workload Increases (Suspend) (2 critical sections per task, 2 lockers per mutex, critical section length 500μs, average of 5 tasks per processor in initial workload)

TABLE I
COST OF PRIMITIVES USED IN RT-MAP

Primitive	Avg. Time (μs)	Max. (μs)
Priority Switching	68.5	433.5
Lock Request	34.6	524.7
Unlock Response	43.2	280.6
Binding Task to Processor Core	2.7	7.5

A basic primitive used in our implementation is that of priority switching. It is required twice per global critical section, once to elevate the task priority and once to restore the task priority to normal. In our implementation, each thread suspends/spins on its own local data-structure and the task releasing the lock is responsible for notifying the appropriate pending thread. The implementation cost is thus evenly distributed between the lock request and the unlock response. Although our implementation does not have the benefits of an in-kernel implementation, the prevalent use of `pthreads` in multi-threaded applications enables us to transparently support a wide-range of applications. Other than the `pthread_setaffinity_np()` for CPU binding, all the other interfaces are expected to be readily portable.

In all our experiments presented in the earlier section, we do not consider any additional preemption costs (caused due to both loss of cache state and context switching overheads). This is due to multiple reasons: (i) we do not wish to bias the results towards our particular implementation, (ii) preemption costs are highly dependent on the particular architecture under consideration - for instance, a multicore system with tightly-coupled memory has an extremely low preemption cost compared to a cache-based multicore processor, and (iii) context-switching costs vary widely across operating system implementations (as low as tens of cycles in ThreadX). Implementation costs, however, cannot always be ignored for a given system. We therefore examined the impact of various preemption costs on the different execution control policies.

Spinning suffers from fewer preemptions compared to the suspension-based scheme. We, therefore, investigated the tipping point at which implementation costs favor *MPCP:Spin* over *MPCP:Suspend*. For each additional job released in the system, both schemes incur an additional preemption cost. This baseline cost, therefore, is assumed to be accounted in job worst-case execution times. *MPCP:Suspend*, however, incurs additional preemptions during each remote-blocking interval.

Our experimental results for various preemption costs are shown in Figure 13, for a workload of 8 fully-packed processors, 2 critical sections per task, and 2 lockers per mutex. For critical sections longer than 40μs, even a 320μs preemption cost does not favor spinning over suspension. On the other hand, at lower preemption costs, *MPCP:Suspend* performs much better. Architecture and application characteristics therefore play a vital role in choosing the execution control policy.

In order to quantify the impact of the preemption overhead on tasks with more critical sections per task, we conducted the experiments summarized in Figure 14. In order to magnify the effect, a 5μs critical section was chosen and two different preemption costs (45μs and 200μs) were considered. The workload was maintained a constant at 8 fully-packed processors and 2 lockers per mutex. As can be seen, with more critical sections per task, the preemption costs tend to significantly affect the suspension-based scheme.

Although we have empirically shown the high preemption costs and low critical section durations for which *MPCP:Spin*

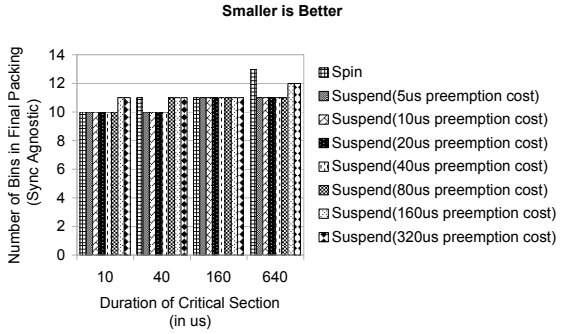

Fig. 13. Impact of Preemption Costs (at different critical section lengths)

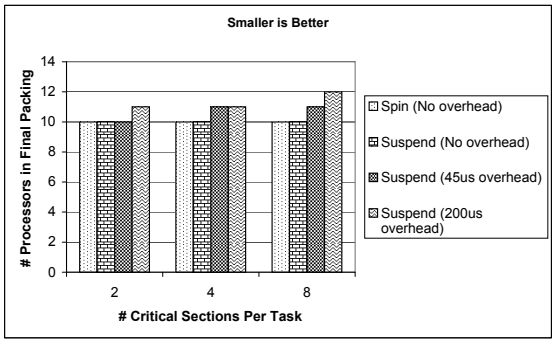

Fig. 14. Impact of Preemption Costs for a 5μs critical section (with increasing number of critical sections per task)

outperforms *MPCP:Suspend*, in practice, such situations are rare. Assuming symmetric read/write cache bandwidths, it is unlikely that a 5μs preemption by a lower priority task's global critical section evicts 45μs worth of cache state. Many application-level critical sections also tend to be much longer than those found at the kernel-level ([19, 20]), especially in massively multicore systems. Spinning is also not a viable candidate for local critical sections, which would require the operating system to treat such scenarios as special cases.

VI. Conclusion

In this paper, we developed a coordinated approach for dealing with task synchronization in chip-multiprocessors. We identified the major scheduling inefficiencies associated with remote task synchronization in fixed-priority multiprocessor scheduling. In order to minimize these scheduling inefficiencies, we analyzed and evaluated different execution control policies. Subsequently, we developed a synchronization-aware task allocation algorithm to explicitly consider these penalties. Experimental evaluation suggests that significant benefits can be achieved using coordinated approaches to task scheduling, allocation and synchronization (in some cases up to 50% reduction in the required number of processing cores). Choosing the appropriate execution control policy (*MPCP:Suspend* or *MPCP:Spin*) based on the system characteristics reduces the penalty due to remote blocking. Synchronization-aware task allocation further reduces the penalty from global resource sharing. Our implementation is available in the form of RT-MAP, a *pthreads*-based library developed for Linux 2.6.22. Future work involves accommodating the cache hierarchy as a part of bin-packing, and evaluating its impact on performance.

VII. Acknowledgements

The authors would like to thank Prof. James Anderson and Bjoern Brandenburg for stimulating discussions on the topic of execution control policies.

References

[1] "Threadx - real-time operating system," http://rtos.com/page/product.php?id=2.
[2] "Autosar - automotive open system architecture," http://www.autosar.org.
[3] J.Lehoczky, L.Sha, and Y.Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behavior," *RTSS*, pp. 166–171, 1989.
[4] L.Sha, R.Rajkumar, and J.P.Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization," *IEEE Trans. on Computers*, vol. 39, no. 9, pp. 1175–1185, 1990.
[5] R.Rajkumar, L.Sha, and J.P.Lehoczky, "Real-time synchronization protocols for multiprocessors," *RTSS*, pp. 259–269, 1988.
[6] R.Rajkumar, "Real-time synchronization protocols for shared memory multiprocessors," *The Tenth International Conference on Distributed Computing Systems*, 1990.
[7] D.S.Johnson, "Near-optimal bin packing algorithms," *Ph.D. Thesis, Department of Mathematics, MIT*, 1973.
[8] J.M.Lopez, J.L.Diaz, and D.F.Garcia, "Minimum and maximum utilization bounds for multiprocessor rm scheduling," *Proceedings of ECRTS*, 2001.
[9] D. de Niz and R. Rajkumar, "Partitioning bin-packing algorithms for distributed real-time systems," *International Journal of Embedded Systems*, vol. 2, pp. 196–208, 2006.
[10] T.P.Baker, "A stack-based resource allocation policy for realtime processes," *RTSS*, pp. 191–200, 1990.
[11] R.Rajkumar, *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.
[12] P.Holman and J.H.Anderson, "Locking in pfair-scheduled multiprocessor systems," *RTSS*, pp. 149–158, 2002.
[13] P.Gai, M. Natale, G.Lipari, A.Ferrari, C.Gabellini, and P.Marceca, "A comparison of mpcp and msrp when sharing resources in the janus multiple-processor on a chip platform," *RTAS*, pp. 189–198, 2003.
[14] B.B.Brandenburg, J.M.Calandrino, A.Block, H.Leontyev, and J.H.Anderson, "Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin?" *RTAS*, pp. 342–353, 2008.
[15] B.B.Brandenburg and J.H.Anderson, "A comparison of the m-pcp, d-pcp, and fmlp on $litmus^{RT}$," *Proceedings of the 12th International Conference on Principles of Distributed Systems*, pp. 105–124, 2008.
[16] L. Ming, "Scheduling of the inter-dependent messages in real-time communication," *Proceedings of the International Workshop on Real-Time Computing Systems and Applications*, 1994.
[17] R.Rajkumar, *Dealing with Suspending Periodic Tasks*. IBM Thomas J. Watson Research Center, Yorktown Heights, 1991.
[18] "Posix threads programming," http://www.llnl.gov/computing/tutorials/pthreads/.
[19] J. Lee and K.H.Park, "The lock hold time prediction of non-preemptible sections in linux 2.6.19," *TR, KAIST*, 2007.
[20] J. Lee and K. H. Park, "Delayed locking technique for improving real-time performance of embedded linux by prediction of timer interrupt," *Proceedings of the 11th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2005.