Reducing Queue Lock Pessimism in Multiprocessor Schedulability Analysis

Yang Chang, Robert Davis, Andy Wellings

Reducing Queue Lock Pessimism in Multiprocessor Schedulability Analysis

Yang Chang, Robert I. Davis and Andy J. Wellings
University of York, UK, {yang, robdavis, andy}@cs.york.ac.uk

***Abstract*—Although many multiprocessor resource sharing protocols have been proposed, their impacts on the schedulability of real-time tasks are largely ignored in most of the existing literature. Recently, work has been done to integrate queue locks (FIFO-queue-based non-preemptive spin locks) with multiprocessor schedulability analysis but the techniques used introduce a substantial amount of pessimism. For global fixed task priority preemptive multiprocessor systems, this pessimism impacts low priority tasks, greatly reducing the number of tasksets that can be recognised as schedulable. A new schedulability analysis *lp-CDW* is designed specifically for analyzing low priority tasks much more accurately. However, this analysis cannot retain its accuracy when it is used to analyze high priority tasks. Existing techniques outperform *lp-CDW* in such cases. By combing *lp-CDW* with existing techniques, we get a hybrid analysis, which performs well at all priorities and therefore significantly increases the number of tasksets that can be recognised as schedulable.**

I. Introduction

More and more real-time embedded systems are being built with multiprocessor/multicore technology. This is the industry's response to the physical limitation on processor clock speeds. Motivated by this trend, the real-time embedded systems research community has recently given much attention to extending the knowledge gained in the uniprocessor era to the development of real-time multiprocessor systems. Most of the techniques developed for uniprocessor systems cannot simply be reused in a multiprocessor environment. Schedulability analysis and resource sharing are prominent examples of such transition difficulties.

Schedulability analysis for uniprocessor systems has been studied for decades and is well understood. Exact analyses (both sufficient and necessary) have been developed for tasks that share resources. In order to bound and reduce each task's *worst-case blocking time*, resource sharing protocols have been proposed, among which the *Priority Inheritance Protocol*, *Priority Ceiling Protocol* [1] and *Stack Resource Policy* [2] are the most widely used.

The most thoroughly studied scheduling policies for real-time tasks on identical multiprocessors are *fully partitioned* and *global* scheduling policies [3]. A fully partitioned system allocates each task to a single processor and disallows any task migration. This divides the multiprocessor scheduling problem into a task allocation problem and a uniprocessor scheduling problem. By contrast, a global system dynamically determines on which processor a task should be executed. Execution of a job may migrate from one processor to another. Multiprocessor real-time scheduling can also be categorised, according to when the priorities can be changed, into: *fixed task-priority*, *fixed job-priority* and *dynamic-priority*. This work focuses on global fixed task-priority preemptive scheduling (*global FP scheduling*).

Schedulability of global FP systems has been studied and many analysis approaches have been proposed. However, global FP schedulability analyses are not as mature as their uniprocessor counterparts. Tractable exact analysis is so far unknown for sporadic tasks, and resource sharing is ignored in most existing work; although efforts have been made on multiprocessor resource sharing protocols themselves. These efforts have resulted in the *Multiprocessor Priority Ceiling Protocol (MPCP)* [4], *Multiprocessor Stack Resource Policy (MSRP)* [5] and the *Flexible Multiprocessor Locking Protocol (FMLP)* [6]. One of the building blocks of these protocols is the *queue lock* (FIFO-queue-based non-preemptive spin lock), which is a simple, yet efficient mechanism of protecting short critical sections on multiprocessors [7]. In a queue lock system, a task becomes non-preemptible when it tries to access a shared resource. If the resource is available, the requesting task locks it and then accesses it non-preemptively. Otherwise, the requesting task busy-waits non-preemptively in a FIFO queue until the resource is made available to it. Eventually the task releases the lock and becomes preemptible again.

In this paper, we focus on the impact of non-nested queue locks on global FP multiprocessor schedulability analysis. The reasoning and results of this study can also be easily applied to systems where nested resource accesses are protected by a shared *group lock* [6].

Even this simple mechanism's impact on multiprocessor schedulability analysis has not been well understood. The state-of-the-art approach to modeling queue locks inflates the worst-case execution time of every task to account for the longest time that could be spent on spinning by that task [4], [5], [6], [8]. *For a global FP system, this introduces too much pessimism at low priority levels where more tasks can interfere with the task under analysis.* The main contribution of this paper is to provide a new model for analyzing queue locks that significantly reduces this pessimism at low priorities. By supplementing an existing approach (designed for independent tasks) with this new model, we obtain a new sufficient analysis *lp-CDW* (in which "lp" stands for low

This work has been funded by the European Commission FP7 JEOPARD and eMuCo projects and the EPSRC Tempo project (EP/G055548/1)

priority and "CDW" is the initials of the authors' names). Experiments reveal that *lp-CDW* is much less pessimistic than the state-of-the-art approach when low priority tasks are being analyzed [9]. However, the state-of-the-art approach works better at high priorities. As will be seen in section VI, combining both analyses can significantly increase the number of tasksets that can be recognised as schedulable.

This paper is organized as follows. First, we summarise existing work in section II. Next, section III presents our task model, terminology and notation. This is followed by discussion of the source of pessimism in existing work and the introduction of a new approach to modeling spinning time. Section V elaborates on the proposed *lp-CDW* analysis. Experimental results are presented in section VI. Finally, we draw conclusions and discuss possible future work.

II. Related Work

A. Multiprocessor Resource Sharing

Because of the discovery of the "Dhall effect" by Dhall and Liu [10], global multiprocessor scheduling was, for many years, considered inferior to the fully partitioned approach and therefore initial efforts on multiprocessor resource sharing protocols mainly focused on fully partitioned systems. This has resulted in two approaches: MPCP [4] and MSRP [5]. MPCP and MSRP both prevent deadlocks and bound the worst-case blocking time of each task as a function of other tasks' critical section lengths rather than other tasks' execution times. However, when a task is denied access to a global resource, MPCP suspends this task and allows lower priority local tasks to execute (and even lock resources) while MSRP works according to the queue lock algorithm.

Recently, more attention has been given to resource sharing protocols for globally scheduled multiprocessors. Block et al. [6] proposed a new policy called FMLP, which categorises resources into two classes: *short* and *long*. The queue lock algorithm is used to protect accesses to short resources. On the other hand, long resources are protected by suspension-based locks with priority inheritance. The only requirement regarding nested resource accesses is that no long resource access can be nested within a short resource access. Easwaran and Andersson [11] proposed another protocol called *parallel-PCP* or *P-PCP*. This is a generalization of uniprocessor PCP for global FP multiprocessor systems. Instead of using non-preemptive spin locks, P-PCP suspends tasks when resources cannot be accessed. A unique feature of this protocol is a new mechanism that limits the system-wide parallelism of resource accesses.

To the best of our knowledge, only a small number of existing global multiprocessor schedulability analysis papers deal with resource sharing [8], [11]. Although resource sharing protocols are usually proposed along with some blocking time analyses, full schedulability analysis is rarely given in these papers [4], [5], [6].

B. Schedulability Analysis for Independent Tasks

Over the last decade, many analyses have been proposed for independent tasks on multiprocessors. In Baker's seminal work [12], an analysis based on the *problem job* and *problem window* was presented for both global EDF and global FP scheduling. Both Baruah [13] and Bertogna and Lipari [14] noticed the pessimism of Baker's analysis and subsequently proposed their own improvements.

Baruah's analysis [13] limits the number of tasks that can carry in any workload (into the problem window) by setting the beginning of each problem window to the last time instant before its problem job's arrival when any processor can be idle. By contrast, Bertogna and Lipari's analyses [14] assume each problem window starts at the same time as its problem job's arrival. In the iterative version of their analyses, the slack of each task is considered when determining its carry-in contribution to the total interference to the problem job. This analysis has recently been improved upon by Guan et al. [15]. Alternative analyses that were also inspired by Baker's work include [16], [17].

III. Model, Terminology and Notation

In this paper, we focus on global FP scheduling of applications that require resource sharing on a homogeneous multiprocessor system comprising m identical processors. An application consists of a static number (n) of *tasks* $\tau_1 ... \tau_i ... \tau_n$, each of which has a unique ID and priority i ($1 \leq i \leq n$ where n represents the lowest priority).

We assume that each task gives rise to a potentially infinite sequence of jobs and that all the jobs of a task are released either *periodically* at fixed intervals of time, or *sporadically* after some minimum inter-arrival time has elapsed. Therefore, every task τ_i can be characterised as (C_i, D_i, T_i) where C_i denotes the worst-case execution time of all the jobs of τ_i excluding any time spent on spinning; D_i represents the relative deadline of each job of τ_i and finally T_i denotes the release period or minimum inter-arrival time of τ_i. It is also assumed that all of the tasks have constrained deadlines, i.e. $D_i \leq T_i$. Furthermore, once a job starts to execute it will not voluntarily suspend itself.

Intra-task parallelism is not permitted; hence, at any given time, each job may execute on at most one processor. As a result of preemption and subsequent resumption, a job may migrate from one processor to another. The cost of preemption, migration, and the runtime operation of the scheduler is assumed to be either negligible, or subsumed into the worst-case execution time of each task.

As noted by Block et al. [6], current global scheduling algorithms (including global FP) do not consider non-preemptive sections. Simply running the highest priority tasks on the remaining preemptible processors is not a good solution as it is possible for a task to be blocked whenever other tasks are released or resumed. Instead, Block et al. proposed the concept of a task being linked to a processor.

A task is linked to a processor at time t if this task would have been scheduled on that processor at time t under the assumption that all tasks are fully preemptible. If a task is linked yet not scheduled, it is deemed *non-preemptively blocked*. During this blocking, the blocked task may be unlinked but it is not allowed to execute anywhere else.

In this work, we assume a standard global FP algorithm has been modified to implement Block et al.'s scheduling scheme (but not their FMLP protocol). By using this algorithm, a job in our system can only be non-preemptively blocked once at the beginning of its execution [6].

In the proposed analysis, if a job of task τ_k arrives at a_k, it can have 5 different states within $[a_k, a_k + D_k)$: *non-preemptively blocked*, *unlinked*, *busy-waiting*, *executing* and *completed*. Task τ_k is said to be non-preemptively blocked when it is currently among the m highest priority ready tasks but cannot be scheduled to run. Task τ_k is said to be unlinked when it is not among the m highest priority ready tasks (τ_k is always ready within $[a_k, a_k + D_k)$ assuming that it has not completed). Task τ_k is said to be busy-waiting when it is spinning non-preemptively, trying to lock a resource that is currently being used by another task. Task τ_k is executing when it is consuming processor time while not busy-waiting. Finally, task τ_k is schedulable if it always completes before or at its deadline.

According to the state of task τ_k, the window $[a_k, a_k + D_k)$ can be divided into 4 sets of time intervals: Θ_k, Γ_k, Λ_k and Ω_k. These are respectively the collection of all the time intervals (not necessarily contiguous) within $[a_k, a_k + D_k)$ during which the job of τ_k is non-preemptively blocked (Θ_k); unlinked (Γ_k); busy waiting (Λ_k); or executing (Ω_k).

We denote by $\rho = \{\rho_1, ..., \rho_j, ...\rho_l\}$ the set of all the resources in the system. Each resource has a unique ID $1 \leq j \leq l$ where l denotes the number of resources in the system. Without loss of generality, we assume that resource accesses are never nested. *Group locks* proposed in [6] can be used to eliminate this restriction without affecting our analysis. All resources are protected by queue locks.

Let $\rho_{i,k}^{x,j}$ denote task τ_i's kth job's xth access to resource ρ_j and $|\rho_{i,k}^{x,j}|$ denote the execution time of $\rho_{i,k}^{x,j}$ (excluding any spinning). Then, we can represent the longest critical section of task τ_i regarding resource ρ_j as $|\rho_i^j| = \max_{k,x}\{|\rho_{i,k}^{x,j}|\}$. The longest critical section of all tasks regarding resource ρ_j is therefore given by $\eta_j = \max_i\{|\rho_i^j|\}$. The longest critical section of tasks with priorities lower than τ_k is given by $b_k = \max_{\{i,j|\tau_i \in lp(k) \wedge \tau_i \in ac(\rho_j)\}}\{|\rho_i^j|\}$ where $lp(k)$ denotes the set of tasks with priorities lower than τ_k and $ac(\rho_j)$ denotes the set of tasks that access ρ_j. The size of $ac(\rho_j)$ is represented by n_j and we further define $\hat{n}_j = \min(m, n_j)$. We also use ψ_i^j to denote the maximum number of accesses to resource ρ_j by any job of task τ_i.

Let $B_{i,k}$ denote the longest time a job of task τ_i can be non-preemptively blocked by tasks with priorities lower than τ_k. We also denote by β_i the worst-case total time a job of task τ_i accesses resources (this does not include any spinning time).

In order to make this paper easier to understand, we summarise some frequently used notations in table I with a brief explanation of their meaning.

Symbols	Definitions
Θ_k	the set of time intervals during which τ_k is non-preemptively blocked
Γ_k	the set of time intervals during which τ_k is unlinked
Λ_k	the set of time intervals during which τ_k is busy waiting
Ω_k	the set of time intervals during which τ_k is executing
Υ_k	the maximum total resource access time introduced by tasks with priorities lower than τ_k within Γ_k
Π_k	the maximum total amount of time that could be wasted on spinning within problem window $[a_k, a_k + D_k)$
Φ_k	the contribution of the execution of tasks with priorities higher than τ_k to the total interference
Δ_k	the contribution of interval Λ_k to the total interference that has not been considered in Π_k
ψ_i^j	the maximum number of accesses to resource ρ_j by any job of task τ_i
$\Psi_{i,k}^j$	the maximum total number of accesses to resource ρ_j by task τ_i within τ_k's problem window $[a_k, a_k + D_k)$
η_j	the length of the longest critical section of resource ρ_j
$\|\rho_i^j\|$	the length of the longest critical section of resource ρ_j that can be entered by task τ_i
β_i	the worst-case total time a job of task τ_i spends on resource accessing
$\omega_{x,j}$	the total of the x longest $\|\rho_i^j\|$ among all tasks using resource ρ_j
b_k	the length of the longest critical section of any tasks with priorities lower than τ_k
$B_{i,k}$	the longest time a job of task τ_i can be non-preemptively blocked by tasks with priorities lower than τ_k
$G_x^{j,k}$	the maximum number of resource request groups in which x requests to ρ_j can be in parallel within τ_k's problem window $[a_k, a_k + D_k)$

Table I

NOTATION DEFINITION

IV. IMPACT OF QUEUE LOCKS

A. Pessimism in Current Approaches

The state-of-the-art approach to developing queue-lock-aware schedulability analyses is to inflate the worst-case execution time of each task to include time spent on spinning and blocking [5], [6], [8]. All existing studies [5], [6], [8] assume that every resource access protected by a queue lock can be interfered with by $\hat{n}_j - 1$ accesses to the same resource on other processors (Recall that $\hat{n}_j = \min(m, n_j)$ and n_j represents the total number of tasks that access resource ρ_j.). According to [5], [6], [8], the inflated worst-case execution time of task τ_i can be represented as:

$$C_i^{wia} = B_{i,i} + C_i + \sum_{\rho_j \in \rho} (\omega_{\hat{n}_j - 1, j} \cdot \psi_i^j) \quad (1)$$

where $B_{i,i}$ denotes the longest blocking time τ_i can suffer; C_i denotes the worst-case execution time of τ_i excluding spinning time; $\omega_{x,j}$ denotes the total of the x longest $|\rho_i^j|$ among all tasks regarding resource ρ_j.

Every task τ_i characterised by (C_i, D_i, T_i) can then be substituted by τ_i^{wia} which is characterised by (C_i^{wia}, D_i, T_i) to form a new taskset. Many global FP analyses designed for independent tasks can be applied to this new taskset without any significant change. If such an analysis considers τ_i^{wia} schedulable, the corresponding task τ_i will be schedulable

as well. However, inflating every task's C_i according to Equation (1) is pessimistic. This is because not all resource requests can be issued in parallel and serial resource requests never cause any spinning among each other (Requests issued by the same task can only happen serially.).

As illustrated in figure 1, suppose a 4 processor system consists of 4 tasks and 1 shared resource. Also assume that each task except task 1 can only request this resource once between t_1 and t_2 even in the worst case. On the other hand, task 1 could access the resource 100 times within the same time interval. Finally, this example assumes each resource access takes exactly 1 time unit. Figure 1a shows that the maximum total time that could be wasted on spinning in this case occurs when one of each task's resource requests is issued simultaneously and task 1's request is the first to be served and task 1 immediately requests again after its previous access is finished. In this worst case, the total wasted time (shown in grey) is 9 time units. By contrast, all existing analyses [5], [6], [8] would give an estimation of 309 (figure 1b).

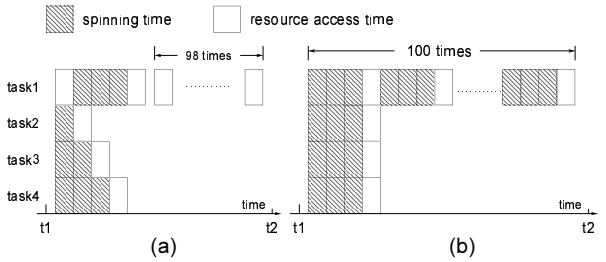

Figure 1. Observed Pessimism

In the current model (under global FP scheduling), when analyzing a task at priority k, only tasks $\tau_i \notin lp(k)$ (where $lp(k)$ denotes the set of tasks with priorities lower than k) have to inflate their worst-case execution times. This is because all $\tau_i \in lp(k)$ have no effect on task τ_k according to this model and hence are not considered when analyzing task τ_k^{wia}. Consequently, if k is a relatively high priority, only a few tasks will have to inflate their worst-case execution times, which counteracts the pessimism of execution time inflation. However, when priority k becomes lower, more and more tasks will have to inflate their execution times and the pessimism increases cumulatively. We therefore expect a performance degradation of the state-of-the-art queue-lock-aware global FP analyses as task priority decreases. Next, we show how to take advantage of our observation to eliminate some pessimism, especially at low priorities.

B. A Less Pessimistic Modeling of Spinning Time

In order to reduce the pessimism cumulated at low priorities in the current model, our new approach groups, for each resource, potentially parallel requests to that resource (issued by tasks at any priority) in each problem window to ignore those that can never be in parallel with others. The worst-case grouping of requests to a specific resource should maximize the total spinning time caused by accesses to that resource in a given problem window. Since the total spinning time consists of the spinning time introduced by tasks with any priority, our new model does not cumulate pessimism as the priority of the problem job decreases. However, when this priority is high, too many tasks (with priorities lower than the problem job) unnecessarily contribute to the estimated total spinning time. Due to the limitation on space, we do not provide proofs for lemmas and theorems given in this paper. They can be found in [9].

First, we consider how resource access requests generate the maximum amount of spinning. In a multiprocessor system that is scheduled according to the FP algorithm of Block et al. described in section III, the maximum amount of spinning time that can be introduced by tasks accessing a resource ρ_j can only be achieved when resource requests on each processor arrive one by one without any delay and the first request in each processor's resource request sequence arrives simultaneously.

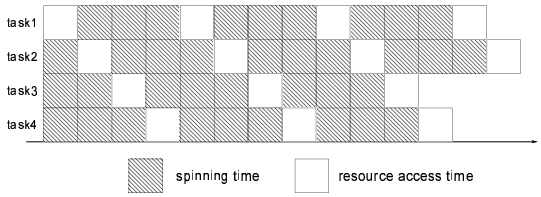

Figure 2. The maximum total spinning time

This situation is illustrated in figure 2 where most of the resource requests are blocked (and therefore spinning) for the same amount of time as described in existing work [5], [6], [8]. However, the observation discussed previously allows opportunities for improvement.

In order to estimate the total spinning time, we need to calculate $\Psi_{i,k}^j$, the maximum total number of accesses to resource ρ_j by task τ_i within τ_k's problem window $[a_k, a_k + D_k)$. This is given below

$$\Psi_{i,k}^j = N_{i,k} \cdot \psi_i^j$$

where $N_{i,k}$ denotes the maximum number of jobs of τ_i that can execute in $[a_k, a_k + D_k)$ and ψ_i^j denotes the maximum number of accesses to resource ρ_j by any job of task τ_i.

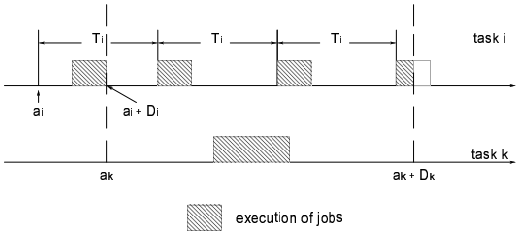

Figure 3. The number of τ_i jobs that can execute in $[a_k, a_k + D_k)$

Based on the worst-case situation given in figure 3, we can derive the following:

$$N_{i,k} = \lceil \frac{D_k + D_i}{T_i} \rceil \quad if \quad i \neq k$$

$$N_{i,k} = 1 \quad if \quad i = k$$

Having got every task's $\Psi_{i,k}^{j}$ regarding a specific resource ρ_j, we could simply assume that any two resource requests can be issued in parallel. However, as discussed previously, not all resource requests can be issued in parallel and those requests that can never be issued in parallel never cause any spinning on each other.

In order to facilitate the proposed total spinning time analysis, we need to group all the accesses to resource ρ_j in $[a_k, a_k + D_k)$ in such a way that each group contains at most $\hat{n}_j$ requests for resource ρ_j issued by different tasks. Because resource requests of the same task can never run in parallel, this grouping method ensures that no unparallelable resource accesses can be in the same group.

Among all the possible results of this grouping method, we are only interested in the worst-case grouping that maximizes the estimated total spinning time in $[a_k, a_k+D_k)$. The development of an algorithm that finds the worst-case grouping requires knowledge of the total spinning time caused by each request group with a different size.

According to the worst case given in figure 4, the maximum total time that could be wasted on spinning by a group of x parallel resource requests (to ρ_j) is $\omega_{x,j} \cdot (x-1)$ where $\omega_{x,j}$ denotes the total of the x longest $|\rho_i^j|$ among all tasks regarding resource ρ_j.

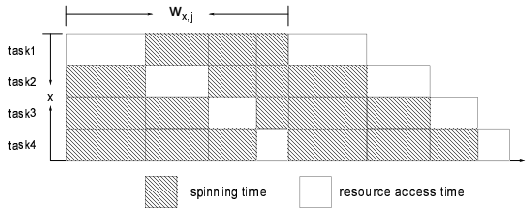

Figure 4. More accurate total spinning time

Lemma 1. *Suppose a taskset either has $\hat{n}_j < 4$, or satisfies the restriction that for any $3 < x \leq \hat{n}_j$, $\omega_{x,j} - \omega_{x-1,j} \geq \frac{x-3}{x-1}(\omega_{x-1,j} - \omega_{x-2,j})$. Then, such a taskset is guaranteed to have the following characteristic for any $2 < x \leq \hat{n}_j$ and $x' = x - 1$:*

$$(x-1)\omega_{x,j} - (x-2)\omega_{x-1,j} \geq (x'-1)\omega_{x',j} - (x'-2)\omega_{x'-1,j} \quad (2)$$

In essence, Lemma 1 suggests that by respecting the above restriction, the total spinning time difference between a size x group and a size $x-1$ group is always no less than that between a size $x-1$ group and a size $x-2$ group.

The restriction described in lemma 1 requires that for any $3 < x \leq \hat{n}_j$, the xth largest $|\rho_i^j|$ among all tasks regarding resource ρ_j should be no less than $\frac{x-3}{x-1}$ times of the $(x-1)$th largest. For those tasksets that do not obey this restriction, we can easily inflate some of the $\hat{n}_j$ largest $|\rho_i^j|$ regarding resource ρ_j when calculating each $\omega_{x,j}$. Then, the maximum total spinning time that could be caused by any group of x resource requests is still $(x-1)\omega_{x,j}$ (with $\omega_{x,j}$ recalculated).

For any taskset (or any adjusted taskset) as described in lemma 1, if we use g_x to represent the number of size x groups, the total spinning time can then be denoted as $\sum_{x=\hat{n}_j}^{2}(x-1)\omega_{x,j} \cdot g_x$. The worst-case grouping should maximize this estimated total spinning time.

Theorem 1. *Suppose there is an algorithm that makes as many size x parallel request groups as possible where x is initially set to $\hat{n}_j$ and decreases x only when the remaining requests can no longer be grouped to the current group size. For any taskset (or any adjusted taskset) as described in Lemma 1, this algorithm gives the worst-case grouping and therefore maximizes the estimated total spinning time $\sum_{x=\hat{n}_j}^{2}(x-1)\omega_{x,j} \cdot g_x$.*

Algorithm 1 Calculate $G_x^{j,k}$ for every $2 \leq x \leq \hat{n}_j$

Input: j, k and non-zero $\Psi_{i,k}^{j}$ of every τ_i.
Output: $G_x^{j,k}$ ($2 \leq x \leq \hat{n}_j$), the maximum number of resource request groups in which x resource requests can be in parallel

```
1: ε = 0;
2: for x = n̂_j to 2 do
3:     G_x^{j,k} = 0;
4: end for
5: loop
6:     Sort Ψ_{i,k}^j in ascending order to form list list;
7:     Let L denote the length of list list;
8:     if L < n̂_j − ε then
9:         ε = ε + 1;
10:        if ε = n̂_j − 1 then
11:            return
12:        end if
13:    else
14:        G^{j,k}_{n̂_j−ε} = G^{j,k}_{n̂_j−ε} + 1;
15:        for each of the last (n̂_j − ε) Ψ_{i,k}^j in list list do
16:            Ψ_{i,k}^j = Ψ_{i,k}^j − 1;
17:            Remove any Ψ_{i,k}^j that becomes zero;
18:        end for
19:    end if
20: end loop
```

Algorithm 1 calculates, for each $2 \leq x \leq \hat{n}_j$ (from $\hat{n}_j$ to 2), $G_x^{j,k}$, the maximum number of resource request groups in which x of the remaining ungrouped requests, can be in parallel.

Line 1 initializes the group size iterator ϵ to zero. $(\hat{n}_j - \epsilon)$ represents the current group size, which is reduced by increasing the group size iterator ϵ. Lines 2-3 initialize the number of groups of every size to zero.

For each iteration of the infinite loop (lines 5-20), we first sort all tasks' non-zero $\Psi_{i,k}^{j}$ in ascending order. If the number of non-zero $\Psi_{i,k}^{j}$ is no smaller than the current group size $(\hat{n}_j - \epsilon)$ (line 8), a new group of size $(\hat{n}_j - \epsilon)$ can be found as a result of this iteration (line 14). In this case, each of the largest $(\hat{n}_j - \epsilon)$ non-zero $\Psi_{i,k}^{j}$ are reduced by one (lines 15 - 18).

When the number of non-zero $\Psi_{i,k}^{j}$ is smaller than the current group size (line 8), it is no longer possible to find any new groups of the current size. Therefore, the group

size is reduced (line 9). If the next group size is one, the algorithm stops (lines 10 and 12).

Because this algorithm always takes requests from the largest x remaining $\Psi^j_{i,k}$ of all tasks (lines 15 - 18) to form a request group of size x, as many tasks' $\Psi^j_{i,k}$ as possible are left greater than zero. Hence, this algorithm is guaranteed to create the biggest possible request group on every iteration. However, this algorithm does not consider different critical section lengths while grouping, which makes Lemma 2 pessimistic. This is necessary because otherwise the complexity of this algorithm would be too high.

Lemma 2. *The maximum total amount of time that could be wasted on spinning by all tasks in $[a_k, a_k + D_k)$ can be upper bounded by:*

$$\Pi_k = \sum_{\rho_j \in \rho} (\sum_{x=\hat{n}_j}^{2} G_x^{j,k} \cdot \omega_{x,j} \cdot (x-1)) \tag{3}$$

V. Schedulability Analysis lp-CDW

The *lp-CDW* analysis is based on Bertogna and Lipari's sufficient non-iterative analysis designed for independent tasks (referred to as *BL*) [14] and requires no further modifications to the standard global FP scheduling apart from those discussed in section III.

The *BL* analysis works on a task by task basis, from the highest priority down to the lowest priority. When analyzing the schedulability of task τ_k, *BL* considers one job of that task a problem job and derives an upper bound on the *interference* of every higher priority task τ_i to the problem job within τ_k's problem window $[a_k, a_k + D_k)$. This interference is defined as the total length of all intervals within $[a_k, a_k + D_k)$ during which τ_k does not execute (though it is ready) while τ_i does. Since the problem job is always ready to execute within $[a_k, a_k + D_k)$, tasks with priorities lower than τ_k can never interfere with the problem job. Moreover, because the global FP scheduling algorithm (without queue locks) is *work conserving* [14], there can never be any idle processor when the problem job does not execute. Therefore, if the sum of the upper bounds on all higher priority tasks' interference to task τ_k is no more than $m(D_k - C_k)$ then all jobs of τ_k will be schedulable.

Compared to the *BL* analysis, the tasksets *lp-CDW* targets have two distinct differences. First, in our tasksets, tasks with priorities lower than task τ_k can also interfere with τ_k within its problem window $[a_k, a_k + D_k)$. This is because parts of the low priority tasks can be executed non-preemptively. Second, some resource accesses can cause non-preemptible spinning, which wastes computation time. Furthermore, because of the non-preemptible sections, our modified global FP scheduling algorithm is no longer work conserving. Therefore, we need a new definition of interference.

Definition 1. *The total interference (I_k) to the problem job (a job of task τ_k) within its problem window $[a_k, a_k + D_k)$ is the total of any idle time, task execution time or spinning time that happens when τ_k is not executing. For the purpose of this paper, τ_k is not considered to be executing when it is spinning.*

Because our total interference includes all the possible idle time that may exist when the problem job does not execute, we get the following schedulability condition even though our scheduling algorithm is not work conserving.

Theorem 2. *If the total interference (I_k) to the problem job (a job of task τ_k) within its problem window $[a_k, a_k + D_k)$ is no more than $m(D_k - C_k)$, task τ_k will be schedulable.*

As discussed in section III, a problem window is composed of 4 time interval sets: Θ_k, Γ_k, Λ_k and Ω_k. Because the problem job executes in Ω_k, this time interval set contributes nothing to the total interference I_k.

First, we model the interference that should be analyzed across Θ_k, Γ_k and Λ_k. This includes interference caused by the execution of tasks with priorities higher than that of τ_k and the interference caused by the spinning of any task (section IV-B). Other interference (i.e. idle time and lower priority task execution) will be discussed later.

As it is very difficult, if not impossible, to estimate the exact total interference to τ_k's problem job, we instead derive an upper bound for each type of interference and then use the sum of these upper bounds as an upper bound on the total interference I_k.

A. Total Workload in the Problem Window

Let's consider the interference caused by the execution of tasks with priorities higher than τ_k. This has been addressed in the *BL* analysis [14]. They used each task's maximum workload during $[a_k, a_k + D_k)$ as an upper bound on each task's maximum interference during $[a_k, a_k + D_k)$ to estimate the schedulability of τ_k.

The maximum workload of task τ_i ($\tau_i \in hp(k)$) within $[a_k, a_k + D_k)$ can be calculated as:

$$W_i(D_k) = N_i(D_k) \cdot C_i + \min(C_i, D_k + D_i - C_i - N_i(D_k) \cdot T_i)$$

where

$$N_i(D_k) = \lfloor \frac{D_k + D_i - C_i}{T_i} \rfloor$$

Based on Bertogna and Lipari's work, we get the following lemma.

Lemma 3. *The contribution of the execution of tasks with priorities higher than τ_k to the total interference is no more than*

$$\Phi_k = \sum_{\tau_i \in hp(k)} \min(W_i(D_k), D_k - C_k) \tag{4}$$

where $hp(k)$ denotes the set of tasks with priorities higher than that of τ_k.

Next, we study each of the 3 time interval sets Θ_k, Γ_k and Λ_k to investigate what contributes to the total interference I_k during each of them.

B. Θ_k — non-preemptively blocked

Block et al. [6] proved that by disallowing the migration of a job that is linked to a processor until it is unlinked, this job can only be non-preemptively blocked once at the beginning of its execution in the absence of any suspension. The maximum length of this non-preemptive blocking is the longest non-preemptible section of all the jobs with lower priorities.

Based on Block et al.'s work, we get the following lemma:

Lemma 4. *The maximum length of the time interval Θ_k is:*

$$B_{k,k} = \max_{\{i,j|\tau_i \in lp(k) \wedge \tau_i \in ac(\rho_j)\}} (\omega_{\hat{n}_j,j}) \quad (5)$$

where $lp(k)$ denotes the set of tasks with priorities lower than task τ_k; $ac(\rho_j)$ denotes the set of tasks that access resource ρ_j and $\omega_{\hat{n}_j,j}$ denotes the total of the $\hat{n}_j$ longest $|\rho_i^j|$ among all tasks using resource ρ_j.

Since τ_k cannot run during Θ_k, it is trivial to prove the following lemma according to Definition 1.

Lemma 5. *The upper bound on Θ_k's contributions to the total interference I_k is $m \cdot B_{k,k}$.*

It should be noticed that we do not make any assumption about the cause of Θ_k's contribution to the total interference. It may be caused by any task other than τ_k. It may simply be idle time.

C. Γ_k — unlinked

Only four types of execution can contribute to the total interference during Γ_k when task τ_k is not one of the m highest priority ready tasks (unlinked). This includes the execution of tasks with priorities higher than τ_k, the spinning of tasks with priorities higher than τ_k, the spinning of tasks with priorities lower than τ_k and finally the non-preemptive resource accesses of tasks with priorities lower than τ_k. In this subsection, we derive only an upper bound on the low priority tasks' non-spinning contribution to the total interference since all other contributions to the total interference are considered elsewhere.

Lemma 6. *During Γ_k, whenever a task $\tau_i \in lp(k)$ is running non-preemptively, a task $\tau_j \in hp(k)$ must be non-preemptively blocked by τ_i where $lp(k)$ ($hp(k)$) denotes the set of tasks with priorities lower (higher) than task τ_k.*

Lemma 7. *During Γ_k, the maximum amount of non-preemptive resource accesses introduced by tasks with priorities lower than τ_k is no more than:*

$$\Upsilon_k = \min(\sum_{i \in hp(k)} \min(\hat{b_{i,k}}, D_k - C_k), \sum_{i \in lp(k)} \min(\hat{\beta_{i,k}}, D_k - C_k)) \quad (6)$$

where $\hat{b_{i,k}}$ denotes the total time that a higher priority task τ_i can be non-preemptively blocked by resource accesses (without any spinning) of tasks with priorities lower than τ_k within $[a_k, a_k + D_k)$; and $\hat{\beta_{i,k}}$ denotes the total time a lower priority task τ_i non-preemptively accesses any resource within $[a_k, a_k + D_k)$.

Next, we demonstrate how to calculate $\hat{b_{i,k}}$ and $\hat{\beta_{i,k}}$. Figures 5 and 6 illustrate the situations in which $\hat{b_{i,k}}$ and $\hat{\beta_{i,k}}$ reach their maximum values respectively. For $\hat{b_{i,k}}$, this happens when b_k and only b_k is completely carried into the problem window $[a_k, a_k + D_k)$ and ends at the deadline of the carry-in job and then, all other jobs of τ_i are blocked by b_k at their arrivals. Hence, the maximum number of complete b_k executions within the problem window $[a_k, a_k + D_k)$ can be calculated as follows:

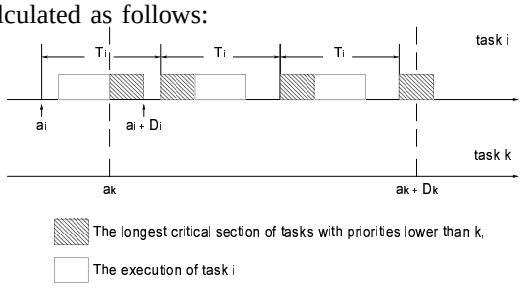

Figure 5. How to calculate $\hat{b_{i,k}}$

$$\tilde{N}_i(k) = \lfloor \frac{D_k + D_i - b_k}{T_i} \rfloor$$

Then, $\hat{b_{i,k}}$ can be represented as:

$$\hat{b_{i,k}} = \tilde{N}_i(k) \cdot b_k + \min(b_k, D_k + D_i - b_k - \tilde{N}_i(k) \cdot T_i)$$

where $\min(b_k, D_k + D_i - b_k - \tilde{N}_i(k) \cdot T_i)$ represents the carry-out part of the non-preemptive resource access.

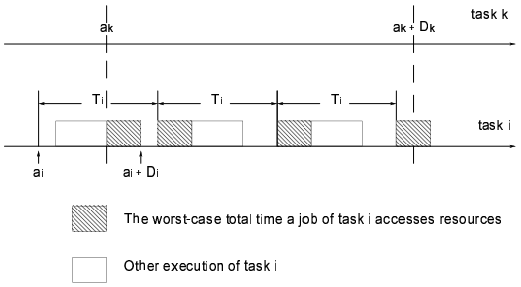

Figure 6. How to calculate $\hat{\beta_{i,k}}$

For $\hat{\beta_{i,k}}$, the worst case happens when τ_i's carry-in job's non-preemptive resource accesses and only those accesses (with a total length of β_i) are carried into the problem window $[a_k, a_k + D_k)$ and the carry-in job completes at its deadline. Then, all other jobs of τ_i run immediately at their arrivals and always make their non-preemptive resource accesses at the beginning. Hence, the maximum number of complete β_i executions within the problem window $[a_k, a_k + D_k)$ is given by:

$$\bar{N}_i(k) = \lfloor \frac{D_k + D_i - \beta_i}{T_i} \rfloor$$

Thus, $\hat{\beta_{i,k}}$ is given by:

$$\hat{\beta_{i,k}} = \bar{N}_i(k) \cdot \beta_i + \min(\beta_i, D_k + D_i - \beta_i - \bar{N}_i(k) \cdot T_i)$$

where $\min(\beta_i, D_k + D_i - \beta_i - \bar{N}_i(k) \cdot T_i)$ represents the carry-out part of the resource accesses.

D. Λ_k — busy waiting

During Λ_k, processors other than τ_k's could be idle or executing any task other than τ_k or spinning waiting for a resource. Irrespective of what these processors are doing, all processors totally contribute $L \cdot m$ to the total interference, where L denotes the maximum length of Λ_k. However, parts of this contribution may have already been considered in the previous subsections. If we let $\Delta_k = L \cdot m$ represent the maximum contribution to the total interference during Λ_k, significant pessimism may be introduced to our analysis. In this subsection, we demonstrate how to improve upon this value of Δ_k.

According to the definition of Λ_k, task τ_k must be spinning waiting for a resource that has been locked by another task. During Λ_k, it is likely that some other tasks are also spinning waiting for the same resource and their requests for this shared resource are queued before τ_k. As all possible spinning time has been considered in Π_k, ignoring some spinning time during Λ_k may prove useful in calculating a less pessimistic value for Δ_k.

First of all, for a request by τ_k for resource ρ_j that is blocked by some other task, suppose that it is blocked by x resource accesses. The longest duration of this blocking is $\omega_{x,j}$ and hence the maximum contribution to the total interference during this blocking is $m \cdot \omega_{x,j}$. In order to make this blocking last $\omega_{x,j}$ time units, the total amount of computation time wasted on spinning by all processors during this blocking must be at least $x \cdot \omega_{x,j} - \sum_{y=1}^{x-1} \omega_{y,j}$ (grey area in figure 7). As the minimum total spinning time during this blocking $x \cdot \omega_{x,j} - \sum_{y=1}^{x-1} \omega_{y,j}$ must have already been considered in Π_k and all x resource accesses within this time interval must have also been considered in Φ_k, we only need to consider $(m-1) \cdot \omega_{x,j} - x\omega_{x,j} + \sum_{y=1}^{x-1} \omega_{y,j} = (m-1-x) \cdot \omega_{x,j} + \sum_{y=1}^{x-1} \omega_{y,j}$ when estimating the contribution to the total interference during this time interval.

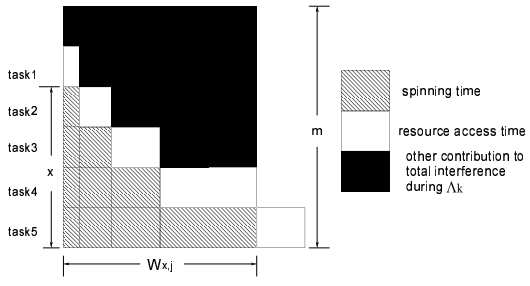

Figure 7. How to calculate Δ_k

It is straightforward to prove:

$$(m-1-x) \cdot \omega_{x,j} + \sum_{y=1}^{x-1} \omega_{y,j} \leq (m-1) \cdot x \cdot \eta_j - \frac{(x+1)^2 - (x+1)}{2} \cdot \eta_j \tag{7}$$

where $(m-1) \cdot x \cdot \eta_j - \frac{(x+1)^2-(x+1)}{2} \cdot \eta_j$ reaches its maximum value $\frac{m^2-3m+2}{2} \cdot \eta_j$ when $x = m-1$. Therefore, we get the following lemma:

Lemma 8. *During Λ_k, the contribution to the total interference that needs to be considered in our analysis is no more than:*

$$\Delta_k = \sum_{\rho_j \in \rho} (\psi_k^j \cdot \frac{m^2 - 3m + 2}{2} \cdot \eta_j) \tag{8}$$

Since the problem job of task τ_k executes in time interval Ω_k, nothing contributes to the total interference to the problem job in this time interval. Therefore, summing the terms in Lemmas 2, 3, 5, 7 and 8, gives an upper bound on the total interference to the problem job (a job of task τ_k) during its problem window $[a_k, a_k + D_k)$:

$$I_k \leq m \cdot B_{k,k} + \Upsilon_k + \Pi_k + \Delta_k + \Phi_k \tag{9}$$

Applying Theorem 2 gives theorem 3.

Theorem 3. *A taskset τ is schedulable on a multiprocessor system with resources shared among tasks according to the queue lock algorithm if for each $\tau_k \in \tau$,*

$$m \cdot B_{k,k} + \Upsilon_k + \Pi_k + \Delta_k + \Phi_k \leq m(D_k - C_k) \tag{10}$$

Because this analysis is based on the new spinning time modeling approach introduced in section IV-B, it reduces pessimism when analyzing low priority tasks compared with the existing analyses using execution time inflation. However, *lp-CDW* performs poorly at high priorities [9] because it always assumes that all tasks contribute to the total spinning time. By contrast, existing analysis works well for high priority tasks since only a few tasks need to inflate their execution times in such cases. A detailed comparison can be found in [9]. In order to make the comparison fair, we use the *BL* analysis as the basis of both *lp-CDW* and the example of the execution time inflation approach. The existing analysis example is a simple use of the *BL* analysis on tasksets modified according to section IV-A. This analysis is referred to as *WIA* in the rest of this paper.

Therefore, it is necessary to combine existing analysis with *lp-CDW* to obtain good results at all priorities. The combination of *WIA* and *lp-CDW* (referred to as *m-CDW* where "m" stands for mixed) analyzes the schedulability of tasks one by one in descending order of their priorities. At each priority, it first uses the *WIA* analysis and if it fails, then the *lp-CDW* analysis is used to check schedulability. Accordingly, algorithm *m-CDW* dominates both *WIA* and *lp-CDW*. As will be seen in section VI and [9], in many experiments, *m-CDW* significantly outperforms both analyses of which it is composed.

VI. Evaluations

In this section, we empirically compare the performance of *WIA*, *lp-CDW* and *m-CDW*. Because Bertogna and Li-

pari's *BL* analysis [14] does not consider any resource sharing and all other analyses are derived from it, *BL* dominates all other analyses discussed in this paper. Therefore, we use the performance of *BL* as a reference for the evaluation of other analyses. Note that due to space limitations, only a few representative results are shown in this paper. See [9] for a comprehensive evaluation. First of all, we present the details of the experiment setup.

A. Methodology

Our experiments were conducted on randomly generated tasksets with variable parameters. Such parameters include the total number of processors m, the priority assignment policy, the total utilization of each taskset, the number of tasks in each set n, the maximum number of resource accesses in any job of any task ψ^{bound} as well as the upper (CS^{ub}) and lower (CS^{lb}) bounds on the randomly generated longest critical section of every resource accessing task.

All experiments assume that only one resource exists. The number of resource accesses ψ_i^j (where j is constant) in any job of task τ_i is randomly generated between 0 and ψ^{bound}. This process is also subject to another restriction, which requires $\sum_i \psi_i^j = \frac{\psi^{bound} \cdot 2n}{m}$. This restriction is introduced to reasonably constrain the experiments and to make different tasksets comparable.

The longest critical section of task τ_i, which is denoted as $|\rho_i^j|$ (where j is constant), has a uniform distribution between CS^{ub} and CS^{lb}. Suppose $\beta^{ub} = |\rho_i^j| \cdot \psi_i^j$. Then, the worst-case total time a job of task τ_i spends on resource accessing, denoted as β_i, has a uniform distribution between $(\beta^{ub} - |\rho_i^j|) \cdot 0.4 + |\rho_i^j|$ and β^{ub}. The coefficient 0.4 is configurable and it controls how close β_i is to β^{ub}.

The total utilization of our tasksets ranges between 0 and m. Task periods in each set have a log-uniform distribution between 2000 and 25000. The utilisation and hence worst-case execution time of each task is generated according to *UUnifast-Discard* [18]. Deadlines have a uniform distribution between the worst case execution times and the periods.

For each experiment, we randomly generate 20000 tasksets for each configuration (including all the above parameters) and record the number of these tasksets that are deemed schedulable by each analysis.

B. WIA *vs* lp-CDW *vs* m-CDW

First, a $m = 4$ processor system is evaluated. The number of tasks in each taskset is set to 25. ψ^{bound}, CS^{lb} and CS^{ub} are set to 5, 10 and 25 respectively.

Figure 8 depicts the performance of *WIA*, *lp-CDW* and *m-CDW* under the DkC priority assignment policy [19]. The x axis in this figure denotes the total utilisation of each taskset. The y axis shows the success rate of each analysis (which is the number of tasksets deemed schedulable by an analysis divided by the total number of tested tasksets, i.e. 20000 in our experiments.). As can be seen in this figure, at

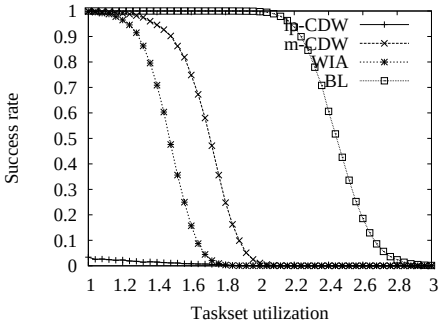

Figure 8. *m-CDW* vs *WIA* on 4 processors when task number is 25

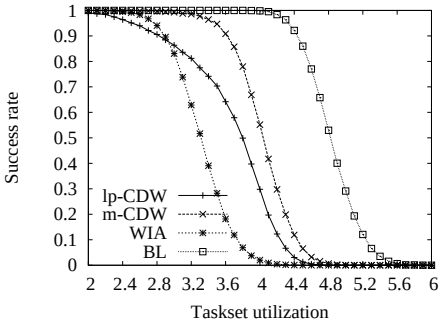

Figure 9. *m-CDW* vs *WIA* on 8 processors when task number is 25

utilisation 1.6, *m-CDW* has a success rate around 75% while the *WIA* analysis can only recognise around 15% of all the randomly generated tasksets as schedulable. Interestingly, the relative performance of *m-CDW* compared against *WIA* remains very good even when *lp-CDW* can barely recognise any schedulable tasksets by itself.

Next, we change the number of processors to $m = 8$ and keep all other parameters unmodified. As illustrated by figure 9, the performance gap between *m-CDW* and *WIA* grows larger and *lp-CDW* improves a lot in this case.

In the next experiment, we study, in more detail, the impact of taskset size on the performance of each schedulability analysis. First, we assume tasks execute on a $m = 4$ processor system (figure 10). The number of tasks in each taskset is varied between 10 and 25. The total utilisation of each taskset tested in this experiment is fixed at 2.0. CS^{lb} and CS^{ub} are set to 5 and 20 respectively. In order to prevent any change to the total utilisation dedicated to resource accesses, this experiment sets $\sum_i \psi_i^j = \frac{\psi^{bound} \cdot 30}{m}$ and $\psi^{bound} = 5$.

The x axis in figure 10 denotes the number of tasks in each taskset. The y axis shows the success rate of each analysis. Figure 10 clearly shows the performance gap between *m-CDW*/*lp-CDW* and *WIA* grows as taskset size increases. Next, we change the number of processors to $m = 8$ and adjust the total utilisations of tasksets to 4. This experiment also sets $\sum_i \psi_i^j = \frac{\psi^{bound} \cdot 30}{m}$ but it changes ψ^{bound} to 10. As illustrated by figure 11, these changes show a similar performance gap between *m-CDW*/*lp-CDW* and *WIA* as observed in the previous experiment.

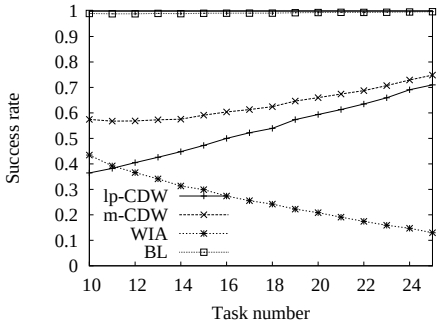

Figure 10. *m-CDW* vs *WIA* when $m = 4$ and total utilisation is 2.0

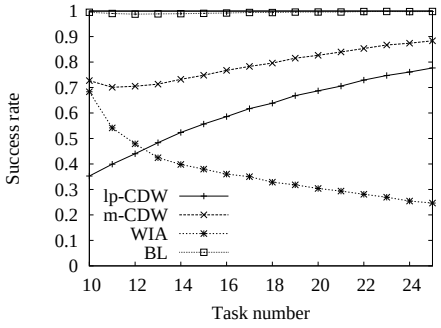

Figure 11. *m-CDW* vs *WIA* when $m = 8$ and total utilisation is 4.0

VII. Conclusions and Future Work

In this paper, we provided a significantly improved schedulability analysis for multiprocessor real-time systems that allow resources to be shared among tasks. This effort was made in particular for global FP multiprocessor scheduling that requires the use of queue locks to protect shared resources. Although queue lock is a very simple resource sharing protocol, it is effective and efficient for many practical industrial applications [7].

To the best of our knowledge, all previous multiprocessor schedulability analyses that consider the use of queue locks take the worst-case execution time inflation approach to modeling time wasted on spinning. It has been shown in this paper how this approach introduces a significant amount of pessimism, especially at low priorities. This motivated the development of a new approach to modeling spinning. It ignores those resource accesses that can never run in parallel with others. Because this new analysis, *lp-CDW*, is designed specifically to eliminate spinning related pessimism at low priorities, it needs to be combined with existing analysis based on worst-case execution time inflation to achieve good performance at all priority levels.

It remains an open question how to apply the intuition behind this work to resource sharing protocols that combine queue locks and suspension-based locks (e.g. *FMLP*). Another issue we intend to solve is nested resource accesses. So far, they are either disallowed in the proposed analyses or assumed to share a common lock.

References

[1] L. Sha, R. Rajkumar, and J. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronisation," *IEEE ToC*, vol. 39, no. 9, pp. 1175–1185, 1990.

[2] T. Baker, "Stack-based scheduling of realtime processes," *Real-Time Systems*, vol. 3, no. 1, pp. 67–99, 1991.

[3] R. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Computing Surveys to appear*, preliminary version available from http://www-users.cs.york.ac.uk/∼robdavis/.

[4] R. Rajkumar, L. Sha, and J. Lehoczky, "Real-time synchronization protocols for multiprocessors," in *Proceedings of Real-time Systems Symposium*, 1988, pp. 259–269.

[5] P. Gai, G. Lipari., and M. Natale, "Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip," in *Proceedings of Real-time Systems Symposium*, 2001, pp. 73–83.

[6] A. Block, H. Leontyev, B. Brandenburg, and J. Anderson, "A flexible real-time locking protocol for multiprocessors," in *Proceedings of RTCSA*, 2007, pp. 47–56.

[7] B. Brandenburg, J. Calandrino, A. Block, H. Leontyev, and J. Anderson, "Real-time synchronization on multiprocessors: To block or not to block, to suspend or spin?" in *Proceedings of RTAS*, 2008, pp. 342–353.

[8] U. Devi, H. Leontyev, and J. Anderson, "Efficient synchronization under global EDF scheduling on multiprocessors," in *Proceedings of ECRTS*, 2006, pp. 75–84.

[9] Y. Chang, R. Davis, and A. Wellings, "Improved schedulability analysis for multiprocessor systems with resource sharing," University of York, Tech. Rep. YCS-2010-454, 2010, http://www.cs.york.ac.uk/ftpdir/reports/2010/YCS/454/YCS-2010-454.pdf.

[10] S. Dhall and C. Liu, "On a real-time scheduling problem," *Operations Research*, vol. 26, no. 1, pp. 127–140, 1978.

[11] A. Easwaran and B. Andersson, "Resource sharing in global fixed-priority preemptive multiprocessor scheduling," in *Proceedings of RTSS*, 2009, pp. 377–386.

[12] T. P. Baker, "Multiprocessor EDF and deadline monotonic schedulability analysis," in *Proceedings of the 24th IEEE Real-time Systems Symposium*, 2003, pp. 120–129.

[13] S. Baruah, "Techniques for multiprocessor global schedulability analysis," in *Proceedings of the 28th IEEE Real-Time Systems Symposium*, 2007, pp. 119–128.

[14] M. Bertogna, M. Cirinei, and G. Lipari, "Schedulability analysis of global scheduling algorithms on multiprocessor platforms," *IEEE Transactions on Parallel and Distributed System*, vol. 20, no. 4, pp. 553–566, 2009.

[15] N. Guan, M. Stigge, W. Yi, and G. Yu, "New response time bounds for fixed priority multiprocessor scheduling," in *Proceedings of RTSS*, 2009, pp. 387–397.

[16] S. Baruah and N. Fisher, "Global fixed-priority scheduling of arbitrary-deadline sporadic task systems," in *Proceedings of the 9th ICDCN*, 2008, pp. 215–226.

[17] S. Baruah and J. Goossens, "The EDF scheduling of sporadic task systems on uniform multiprocessors," in *Proceedings of the 29th RTSS*, 2008, pp. 367–374.

[18] R. Davis and A. Burns, "Priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems," in *Proceedings of RTSS*, 2009, pp. 398–409.

[19] B. Andersson and J. Jonsson, "Fixed-priority preemptive multiprocessor scheduling: to partition or not to partition," in *Proceedings of the RTCSA*, 2000.