Improved Schedulability Analysis for Multiprocessor Systems with Resource Sharing

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

***Abstract*—This report presents our recent efforts to close the gap between the state-of-the-art homogeneous (or identical) multiprocessor and uniprocessor schedulability analyses in the context of resource sharing. Although many multiprocessor resource sharing protocols have been proposed, their impact on the schedulability of real-time tasks is largely ignored in most 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, some of which, as explained in this report, can be easily eliminated. 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. We develop a new schedulability analysis *lp-CDW* to target this issue specifically. By combing *lp-CDW* with existing techniques, we significantly increase the number of tasksets that can be recognised as schedulable.**

I. Introduction

Today, more and more real-time embedded systems are being built with multiprocessor (or multicore) technology. This is the industry's response to the physical limitation on processor clock speeds. Motivated by this trend, the real-time embedded system research community has recently given much attention to extending the knowledge gained in the uniprocessor era to the development of real-time multiprocessor systems [1]. Unfortunately, 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 and task characteristics such as *release jitter*, *constrained/arbitrary deadlines*, *context switch times* etc. have all been considered. One of the task characteristics that has to be considered in any useful schedulability analysis is the *worst-case blocking time* of each task, which represents the maximum total time during which a job of this task is blocked by any lower priority job. In order to bound and reduce the worst-case blocking time as well as to prevent deadlocks, resource sharing protocols have been proposed, among which the *Priority Inheritance Protocol*, *Priority Ceiling Protocol* [2] and *Stack Resource Policy* [3] are the most widely used.

This work has been funded by the European Commission's 7th framework program's JEOPARD project, no. 216682 and eMuCo project, no. FP7-ICT-2007-1

The most thoroughly studied scheduling policies of real-time tasks on homogeneous (or identical) multiprocessors are *fully partitioned* and *global* scheduling policies. 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 3 classes [4] : *fixed task-priority*, *fixed job-priority* and dynamic-priority. This work focuses on global fixed task-priority scheduling, which is referred to as *global FP scheduling* in this report.

Schedulability of global FP scheduling systems has been studied and many analysis approaches have been proposed [5], [6], [7], [8], [9], [10], [11], [12]. 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 though efforts have been made on multiprocessor resource sharing protocols themselves. These efforts have resulted in the *Multiprocessor Priority Ceiling Protocol (MPCP)* [13], *Multiprocessor Stack Resource Policy (MSRP)* [14] and the *Flexible Multiprocessor Locking Protocol (FMLP)* [15]. One of the building blocks of these protocols is *queue lock* (FIFO-queue-based non-preemptive spin lock), which is deemed to be a simple, yet efficient mechanism of protecting short critical sections on multiprocessors [16]. 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. Eventually this task releases the lock and becomes preemptible again. Otherwise, the requesting task busy-waits non-preemptively in a FIFO queue until the resource is made available to it.

There are other spin lock algorithms that make use of prioritised queues, preemptive execution and/or priority inheritance [17], [18]. However, in this report, we focus on

the impact of non-nestable queue locks (i.e. queue locks that do not consider any nested resource access) on global FP multiprocessor schedulability analysis, leaving the study of other spin lock algorithms and nested resource accesses to future work. The essential ideas and improvements proposed in this report may also be applied to other spin lock algorithms.

As will be discussed shortly, 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 [13], [14], [15], [19]. As will be explained shortly, this approach is pessimistic because in reality only some resource accesses spin and they do not always spin for the longest possible duration. For a global FP system, the current worst-case execution time inflation approach 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. 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 priority and "CDW" is the initials of the authors' names). Experiments reveal that combining *lp-CDW* with a state-of-the-art queue-lock-aware analysis can significantly increase the number of tasksets that can be recognised as schedulable.

This report is organized as follows. First, we summarise relevant 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 our new approach to modeling spinning time. Sections V elaborates the proposed *lp-CDW* analyses. In section VI, we discuss why *lp-CDW* should be combined with a state-of-the-art queue-lock-aware analysis and also how we this can be achieved. Experimental results are presented in section VII. Finally, we draw conclusions and discuss possible future work.

II. Related Work

A. Resource Sharing

In 1988, uniprocessor priority inheritance and priority ceiling protocols were proposed by Rajkumar et al. [13] and first integrated with Liu and Layland's utilisation-based analysis [20]. A few year later, Audsley et al. integrated Rajkumar et al.'s protocols with response time analysis [21]. In 1991, Baker [3] proposed another resource sharing protocol called the Stack Resource Policy, which provides simpler support for EDF scheduling and guarantees a job can only be blocked at the beginning of its execution. This makes it possible to use a single shared runtime stack for all the tasks.

Because of the discovery of the "Dhall effect" by Dhall and Liu [22], 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 [13] and MSRP [14]. 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. Moreover, local resource accesses can be nested and are managed according to PCP (SRP) under MPCP (MSRP). Accesses to global resources are not allowed to be nested within any resource access and *vice versa*.

In 1997, Philips et al. [23] showed that the "Dhall effect" was more of a problem with high utilisation tasks than it was with global scheduling algorithms. This discovery not only influenced the schedulability analysis research but may also have affected the course of research in multiprocessor resource sharing.

Recently, more attention has been given to resource sharing protocols for globally scheduled multiprocessors. Block et al. [15] 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 [24] proposed a new protocol called *parallel-PCP* or *P-PCP*. This is a generalization of the 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) schedulability analysis papers deal with resource sharing [19], [24]. Although resource sharing protocols are usually proposed along with some blocking time analyses, full schedulability analysis is rarely given in these papers [13], [14], [15].

B. Multiprocessor Schedulability Analysis for Independent Tasks

Multiprocessor real-time scheduling has attracted much attention during the last decade [1]. Many algorithms and

schedulability analyses have been proposed for independent tasks (i.e. resources are not shared among tasks). Carpenter et al. categorized and compared different types of algorithms and analyses in [4].

In Baker's 2003 seminal work [8], an analysis based on the *problem job* and *problem window* was presented for both global EDF and global FP scheduling. In this approach, a job called the problem job is assumed to be the first one in the whole system to miss a deadline. The problem window of a problem job starts from a time instant before the release of the problem job and ends at the problem job's deadline. The essential idea of this analysis is to establish a condition necessary for the problem job to miss its deadline. This requires the upper bound on the maximum interference caused by other tasks in the problem window.

Both Baruah [9] and Bertogna and Lipari [10] noticed a fact ignored by Baker, that a task's interference to the problem job within its problem window is bounded by the length of the problem window less the problem job's worst-case execution time. They both eliminated this overestimated interference by setting an upper bound on each task's interference to the problem job. They also both noticed another source of pessimism in Baker's analysis, which is the carry-in contribution (workload carried into the problem window by tasks released before the problem window) to the total interference. Baruah's analysis [9] limits the number of tasks that can carry in any workload by setting the beginning of each problem window to the last time instant before its problem job's arrival a_k when any processor can be idle. By contrast, Bertogna and Lipari's (non-iterative and iterative) analyses [10] assume each problem window starts at the same time as its problem job's arrival a_k. 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. [11]. Alternative analyses that were also inspired by Baker's work include [7], [12], [25].

III. Model, Terminology and Notation

In this report, 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 1 (n) represents the highest (lowest) priority).

We assume that each task gives rise to a potentially infinite sequence of jobs and 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 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. in [15], 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 *non-preemptively blocked* whenever other tasks are released or resumed. A task is non-preemptively blocked if this task is one of the m highest priority runnable tasks and it is not scheduled because a lower priority task's non-preemptive section has been scheduled.

Instead, Block et al. proposed the concept of a task being linked to a processor. Basically, 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 to be non-preemptively blocked. During this blocking, the blocked task may be unlinked but it is not allowed to execute on any other processor.

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

In the proposed analyses, if a job of 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 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 $(a_k + D_k)$.

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);
- 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. Different instances of the same type of resource are considered different resources. It is assumed that resource accesses are never nested and all resources are protected by queue locks only.

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 can therefore be denoted as $\eta_j = \max_i(|\rho_i^j|)$. The longest critical section of tasks with priorities lower than τ_k can be denoted as $b_k = \max_{\{i,j|\tau_i \in lp(k) \wedge \tau_i \in ac(\rho_j)\}}(|\rho_i^j|)$ 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 report easier to understand, we summarise some frequently used notations in table I with a brief explanation of their meaning.

IV. Impact of Queue Locks

A. Pessimism in Current Approaches

The state-of-the-art approach to integrating queue-lock-based protocols with schedulability analysis is to inflate the worst-case execution time of each task by the maximum amount of computation time that could be wasted on spinning by the corresponding task [15], [19], [14]. Because it is too difficult to predict what other processors are doing when a resource is requested on a specific processor, all existing analyses [15], [19], [14] assume that every resource access protected by a queue lock can be blocked by $\hat{n}_j - 1$ accesses to the same resource on other processors (recall that $\hat{n}_j = \min(n_j, m)$ and n_j represents the total number of tasks that access resource ρ_j). In other words, it is assumed that whenever a resource is requested, this request is always queued at the end of a full FIFO queue. According to the current approach [19], the inflated worst-case execution time of task τ_i can be represented as:

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

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; η_j denotes the longest critical section of resource ρ_j; ψ_i^j denotes the maximum number of accesses to resource ρ_j in any job of task τ_i.

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.).

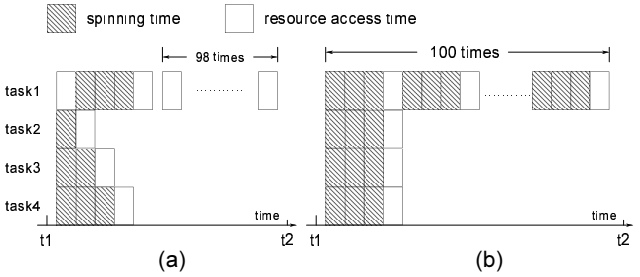

Figure 1. Observed Pessimism

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

Symbols	Definitions
Task Scheduling Related Notations	
m	the total number of processors
n	the total number of tasks in each taskset
n_j	the total number of tasks in each taskset that access resource ρ_j
$\hat{n}_j$	$\hat{n}_j = \min(m, n_j)$
$hp(k)$	the set of tasks with priorities higher than task τ_k
$lp(k)$	the set of tasks with priorities lower than task τ_k
$[a_k, a_k + D_k)$	the problem window of task τ_k
C_i	the worst-case execution time of task τ_i, excluding spinning time
D_i	the relative deadline of task τ_i
T_i	the minimum inter-arrival time or period of task τ_i
Time intervals	
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
Resource Access Number Notations	
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 $[a_k, a_k + D_k)$
$G_x^{j,k}$	the maximum number of resource request groups in which x requests to ρ_j can be in parallel within τ_k's problem window $[a_k, a_k + D_k)$
Resource Access Time Notations (without spinning)	
η_j	the length of the longest critical section of resource ρ_j
$\lvert\rho_i^j\rvert$	the length of the longest critical section of resource ρ_j that can be entered by task τ_i
b_k	the length of the longest critical section of any tasks with priorities lower than τ_k
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 $\lvert\rho_i^j\rvert$ among all tasks using resource ρ_j
Non-Preemptive Section Notations	
$B_{i,k}$	the longest time a job of task τ_i can be non-preemptively blocked by tasks with priorities lower than τ_k
Contributions to Total Interference	
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

Table I

NOTATION DEFINITION

existing analyses [14], [15], [19] would give an estimation of 309 (figure 1b).

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 according to Equation (1). 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 any task) 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. It should be noticed that the total spinning time consists of the spinning time introduced by tasks with any priority. Therefore, our new model does not cumulate pessimism as the priority of the problem job decreases. Because the grouping of requests to a specific resource ignores those requests that can never be in parallel,

this model is more accurate if the priority of the problem job is low. However, when this priority is high, too many tasks (with priorities lower than the problem job) unnecessarily contribute to the estimated total spinning time.

First of all, we consider how resource access requests generate the maximum amount of spinning. In a multi-processor system that is scheduled according to the global 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.

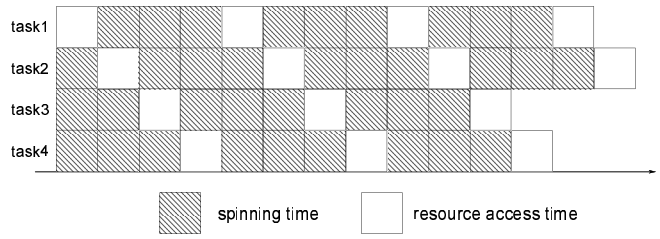

Figure 2. The maximum total spinning time

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 [15], [19], [14].

In order to estimate the total spinning time in the problem window, we need to calculate $\Psi_{i,k}^j$, the maximum total number of accesses to 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 be executed in τ_k's problem window and ψ_i^j denotes the maximum number of accesses to resource ρ_j by any job of task τ_i.

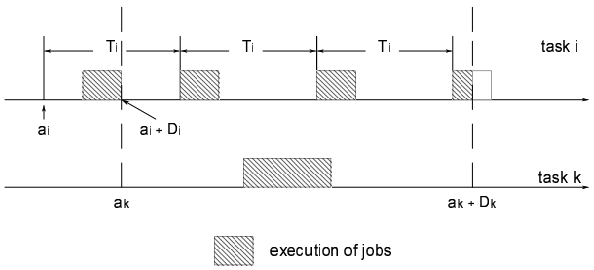

Figure 3. The maximum number of jobs of τ_i that can be executed in τ_k's problem window

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

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

and

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

Having got every task's $\Psi_{i,k}^j$ regarding a specific resource ρ_j in task τ_k's problem window, we could simply assume that any two resource requests (as part of $\sum_i \Psi_{i,k}^j$) can be issued in parallel and then derive a formula based on this assumption to calculate the total spinning time. 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 τ_k's problem window in such a way that each group contains at most m 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 (when used on resource requests in a specific problem window), we are only interested in the worst-case grouping that maximizes the estimated total spinning time in the whole problem window. 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 Block et al. [15] and Gai et al. [14], the longest access to the same resource by different tasks may be different. Therefore, it is pessimistic to assume that each request to resource ρ_j spins for $(\hat{n}_j - 1) \cdot \eta_j$. The real worst case is given in figure 4 according to which 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.

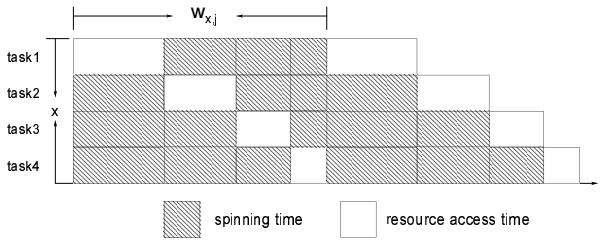

Figure 4. A better estimation of the maximum total spinning time

Then, we get the following lemma.

Lemma 1. *Suppose a taskset either runs on a system that has a $\hat{n}_j < 4$, 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 $x' = x - 1$:*

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

Proof: First, let's assume $x = 3$, then proving the lemma is equivalent to proving that $2\omega_{3,j} - \omega_{2,j} \geq \omega_{2,j}$. According to the definition of $\omega_{x,j}$, $\omega_{3,j} > \omega_{2,j}$ so $2\omega_{3,j} - \omega_{2,j} \geq \omega_{2,j}$.

When $x > 3$, the taskset under discussion must be scheduled on a system with at least $\hat{n}_j = 4$. Therefore, for any $3 < x \leq \hat{n}_j$, $\omega_{x,j} - \omega_{x-1,j} \geq \frac{x-3}{x-1}(\omega_{x-1,j} - \omega_{x-2,j})$ must be satisfied by this taskset. Proving the lemma for this case is equivalent to proving that:

$$(x-1)\omega_{x,j} - (2x-4)\omega_{x-1,j} + (x-3)\omega_{x-2,j} \geq 0 \qquad 3 < x \leq \hat{n}_j$$

Because $\omega_{x,j} - \omega_{x-1,j} \geq \frac{x-3}{x-1}(\omega_{x-1,j} - \omega_{x-2,j})$,

$$(x-1)(\omega_{x,j} - \omega_{x-1,j}) - (x-3)(\omega_{x-1,j} - \omega_{x-2,j}) \geq 0 \qquad 3 < x \leq \hat{n}_j$$

Consequently,

$$(x-1)\omega_{x,j} - (x-1)\omega_{x-1,j} - (x-3)\omega_{x-1,j} + (x-3)\omega_{x-2,j} \geq 0 \qquad 3 < x \leq \hat{n}_j$$

and finally,

$$(x-1)\omega_{x,j} - (2x-4)\omega_{x-1,j} + (x-3)\omega_{x-2,j} \geq 0 \qquad 3 < x \leq \hat{n}_j$$

which proves the lemma.

■

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.

Lemma 2. *Given a taskset as described in Lemma 1, suppose there are integers x_1 and x_2 where $2 < x_1 < x_2 \leq \hat{n}_j$. Then, it is guaranteed that:*

$$(x_2-1)\omega_{x_2,j} - (x_2-2)\omega_{x_2-1,j} \geq (x_1-1)\omega_{x_1,j} - (x_1-2)\omega_{x_1-1,j} \quad (3)$$

Proof: If $x_1 = x_2 - 1$, this is exactly the same as Lemma 1, which has been proved.

Otherwise, if $x_1 < x_2 - 1$, we can recursively apply Lemma 1 to any value between x_1 and $x_2 - 1$ to prove the lemma. ■

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 one. 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 (not necessarily according to the worst-case grouping), 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 always makes as many size x parallel request groups as possible where x is initially set to $\hat{n}_j$ and decreases 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 $\sum_{x=\hat{n}_j}^{2}(x-1)\omega_{x,j} \cdot g_x$.*

This theorem is proved in appendix A.

As an example of such an algorithm, we developed Algorithm 1 to calculate, 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.

In this algorithm, line 1 initializes the group size iterator ϵ to zero. $(\hat{n}_j - \epsilon)$ represents the current group size, which can be reduced by increasing the group size iterator ϵ when necessary. 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 group 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).

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_{n̂_j−ϵ}^{j,k} = G_{n̂_j−ϵ}^{j,k} + 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
```

Because this algorithm always takes requests from the largest x remaining $\Psi_{i,k}^j$ of all tasks (lines 15-18) to form a request group of size x, as many tasks' $\Psi_{i,k}^j$ as possible are always left greater than zero. Hence, this algorithm creates the biggest possible request group on every iteration.

Lemma 3. *The maximum total amount of time that could be wasted on spinning (by any task) in the whole problem window $[a_k, a_k + D_k)$ can be upper bounded by:*

$$\Pi_k = \sum_{\rho_j \in \rho} \Big(\sum_{x=\hat{n}_j}^{2} G_x^{j,k} \cdot \omega_{x,j} \cdot (x-1) \Big) \tag{4}$$

Proof: This lemma is derived from Theorem 1 and $G_x^{j,k}$ represents the value of g_x regarding resource ρ_j according to the worst-case grouping. ■

It should be noticed that Π_k includes all tasks' (even τ_k itself) spinning time that could exist in τ_k's problem window.

V. Analysis LP-CDW

The *lp-CDW* analysis is based on Bertogna and Lipari's polynomial time sufficient non-iterative analysis (referred to as "BL" in this report) [10] and requires no further modifications to the standard scheduling and queue lock algorithm 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 a 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* [10], 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 our analysis targets have two important 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 in this work.

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 report, τ_k is not considered to be executing when it is spinning.*

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.*

Proof: Because our total interference includes all the possible idle time that may exist when the problem job does not execute, we can follow Bertogna and Lipari's work [10] to get the theorem even though our scheduling algorithm is not work conserving. ■

As discussed previously 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. In the other 3 time interval sets, there is interference that can be more accurately analyzed across all time interval sets. There is also interference that is unique to a specific time interval set. Such interference can be better analyzed within its own time interval set.

First, we model the interference that should be analyzed across Θ_k, Γ_k and Λ_k. This includes the 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 [10]. 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

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

Lemma 4. *The contribution of the execution of tasks with priorities higher than τ_k to the total interference is no more than*

$$\Phi_k = \sum_{\tau_i \in hp(k)} \min(W_i(D_k), D_k - C_k) \tag{5}$$

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

Proof: Follows from Bertogna and Lipari's analysis [10]. ■

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. [15] 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.

Lemma 5. *The maximum length of the time interval Θ_k is:*

$$B_{k,k} = \max_{\{i,j|\tau_i \in lp(k) \wedge \tau_i \in ac(\rho_j)\}} (\omega_{\hat{n}_j,j}) \tag{6}$$

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.

Proof: Follows from Block et al.'s work [15]. ■

Lemma 6. *The upper bound on Θ_k's contributions to the total interference I_k is $m \cdot B_{k,k}$.*

Proof: Since τ_k cannot run during Θ_k, all the execution time, spinning time, idle time occurred within this time interval contributes to the total interference I_k. According to Lemma 5, the maximum length of Θ_k is $B_{k,k}$. Hence, 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 on 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

Lemma 7. *During Γ_k, all processors must be running (executing and/or spinning) either tasks with priorities higher than τ_k or the non-preemptible sections of tasks with priorities lower than τ_k.*

Proof: Proving this lemma is equivalent to proving that no processor can be idle or running any pre-emptable section of any task with a priority lower than τ_k during Γ_k.

Suppose there is an idle processor during Γ_k. As there is at least one ready task that has a priority no lower than τ_k and that is currently unlinked during Γ_k, the scheduler would have picked this task to run on the idle processor, which contradicts the assumption. Based on the same reasoning, no pre-emptable sections of any task with a priority lower than τ_k can be scheduled in preference to τ_k which is aways ready during Γ_k. ■

According to this lemma, only four types of execution can contribute to the total interference during Γ_k. 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 8. *During Γ_k, the maximum amount of non-preemptive resource accesses introduced by tasks with priorities lower than τ_k is no more than:*

$$\sum_{i \in lp(k)} \min(\hat{\beta_{i,k}}, D_k - C_k) \tag{7}$$

where $\hat{\beta_{i,k}}$ denotes the total time task τ_i non-preemptively accesses any resource within $[a_k, a_k + D_k)$.

Proof: Because Γ_k is part of the problem window $[a_k, a_k + D_k)$, the maximum amount of non-preemptive resource accesses introduced by task $\tau_i \in lp(k)$ during Γ_k can never be more than $\hat{\beta_{i,k}}$ the total time (without any spinning) task τ_i accesses any resource during $[a_k, a_k + D_k)$. Therefore, during Γ_k, the maximum amount of non-preemptive resource accesses introduced by tasks with priorities lower than τ_k can never be more than $\sum_{i \in lp(k)} \hat{\beta_{i,k}}$. Moreover, if Γ_k is longer than $D_k - C_k$ then the taskset will definitely fail the test. According to Bertogna and Lipari [10], $\sum_{i \in lp(k)} \hat{\beta_{i,k}}$ can be substituted by $\sum_{i \in lp(k)} \min(\hat{\beta_{i,k}}, D_k - C_k)$. ■

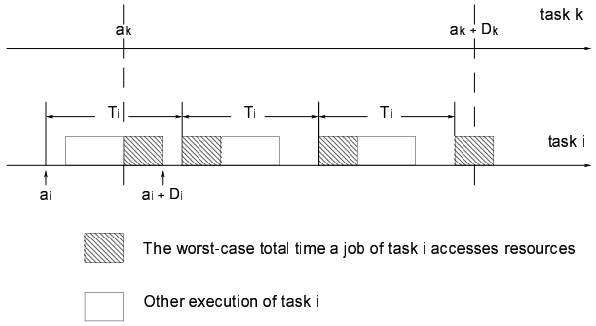

Figure 5. How to calculate $\hat{\beta_{i,k}}$

Next, we demonstrate how to calculate $\hat{\beta_{i,k}}$. Figure 5 illustrates the situation in which $\hat{\beta_{i,k}}$ reaches its maximum value. 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 run all their non-preemptive resource accesses at the beginning. Hence, the maximum number of complete β_i execution 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.

Lemma 9. *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. $lp(k)$ ($hp(k)$) denotes the set of tasks with priorities lower (higher) than task τ_k.*

Proof: Suppose a task τ_i runs non-preemptively without blocking any task with a priority higher than τ_k during Γ_k. Then, τ_k would have been linked to τ_i's processor. This contradicts the definition of Γ_k. ■

According to Lemma 9, whenever a non-preemptive resource access is introduced by a task $\tau_i \in lp(k)$ during Γ_k, this resource access must non-preemptively block a task $\tau_j \in hp(k)$. According to Block et al. [15], a high priority task $\tau_j \in hp(k)$ can only be non-preemptively blocked once per release. Consequently, the maximum number of non-preemptive resource accesses introduced by tasks with priorities lower than τ_k during Γ_k is bounded by the maximum number of releases of tasks with priorities higher than τ_k during the same time interval.

Lemma 10. *During Γ_k, the maximum amount of non-preemptive resource accesses introduced by tasks with priorities lower than τ_k is no more than:*

$$\sum_{i \in hp(k)} \min(\hat{b_{i,k}}, D_k - C_k) \tag{8}$$

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

Proof: According to Block et al. [15], our high priority task $\tau_i \in hp(k)$ can only be non-preemptively blocked once per release at the beginning of its execution in the absence of any suspension. Consequently, in the worst case, every release of $\tau_i \in hp(k)$ during $[a_k, a_k + D_k)$ corresponds to a non-preemptible section of some $\tau_j \in lp(k)$. To make things even worse, it is possible that all such non-preemptible sections are executed during Γ_k. Therefore, the total amount of resource accesses of tasks with priorities lower than τ_k that actually executes within Γ_k and causes task τ_i to be blocked is no more than $\hat{b_{i,k}}$.

Moreover, according to Lemma 9, any non-preemptive execution caused by $\tau_i \in lp(k)$ during Γ_k must block some task with a priority higher than τ_k. Otherwise τ_k would be linked. Therefore, during Γ_k, the maximum amount of

non-preemptive resource accesses introduced by tasks with priorities lower than τ_k is no more than $\sum_{i\in hp(k)} \hat{b_{i,k}}$. Due to the limitation of the length of Γ_k, this upper bound can be improved to $\sum_{i\in hp(k)} \min(\hat{b_{i,k}}, D_k - C_k)$ ■

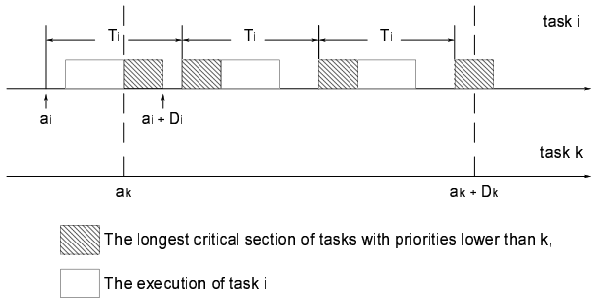

Figure 6. How to calculate $\hat{b_{i,k}}$

Figures 6 illustrate the situations in which $\hat{b_{i,k}}$ reaches its maximum value. This happens when b_k (length of the longest critical section of tasks $\tau_j \in lp(k)$) 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 execution within the problem window $[a_k, a_k + D_k)$ can be calculated as follows:

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

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.

Lemma 11. *During Γ_k, the maximum amount of non-preemptive resource accesses introduced by tasks with priorities lower than τ_k is no more than:*

$$\Upsilon_k = \min(\sum_{i\in hp(k)} \min(\hat{b_{i,k}}, D_k - C_k), \sum_{i\in lp(k)} \min(\hat{\beta_{i,k}}, D_k - C_k)) \quad (9)$$

Proof: Follows from Lemmas 8 and 10. ■

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 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 processors are also spinning waiting for the same resource and their requests for this shared resource are queued before τ_k. This may cause further spinning of 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.

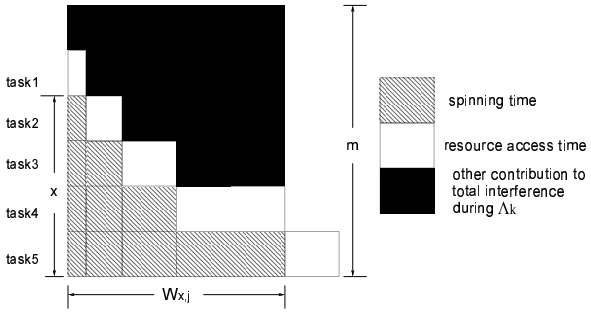

Figure 7. How to calculate Δ_k

Lemma 12. *For any $0 < x \leq m - 1$,*

$$(m-1-x)\cdot\omega_{x,j} + \sum_{y=1}^{x-1} \omega_{y,j} \leq (m-1)\cdot x \cdot \eta_j - \frac{(x+1)^2 - (x+1)}{2} \cdot \eta_j \quad (10)$$

Proof: Suppose that $|\rho_i^j|$ of task τ_i is among the x

largest ones. Then, we define $q_i^j = \eta_j - |\rho_i^j|$. Therefore, if we enlarge the maximum critical section of task τ_i regarding resource ρ_j to η_j, we obtain:

$$\hat{\omega}_{x,j} = \omega_{x,j} + q_i^j \tag{11}$$

where $\hat{\omega}_{x,j}$ is used to denote the $\omega_{x,j}$ after the maximum critical section of task τ_i regarding resource ρ_j is inflated to η_j. Hence, we get:

$$\begin{aligned}(m-1-x)\cdot\hat{\omega}_{x,j} + \sum_{y=1}^{x-1}\hat{\omega}_{y,j} - ((m-1-x)\cdot\omega_{x,j} + \sum_{y=1}^{x-1}\omega_{y,j}) \\ = (m-1-x)q_i^j + (\sum_{y=1}^{x-1}\hat{\omega}_{y,j} - \sum_{y=1}^{x-1}\omega_{y,j})\end{aligned} \tag{12}$$

Because $0 < x \leq m-1$, $(m-1-x)q_i^j \geq 0$. Moreover, because $\hat{\omega}_{x,j} \geq \omega_{x,j}$ for any $0 < x \leq m-1$, $\sum_{y=1}^{x-1}\hat{\omega}_{y,j} - \sum_{y=1}^{x-1}\omega_{y,j} \geq 0$. Therefore,

$$(m-1-x)q_i^j + (\sum_{y=1}^{x-1}\hat{\omega}_{y,j} - \sum_{y=1}^{x-1}\omega_{y,j}) \geq 0 \tag{13}$$

Applying equation 12 gives:

$$(m-1-x)\cdot\hat{\omega}_{x,j} + \sum_{y=1}^{x-1}\hat{\omega}_{y,j} - ((m-1-x)\cdot\omega_{x,j} + \sum_{y=1}^{x-1}\omega_{y,j}) \geq 0 \tag{14}$$

Consequently, the value of $(m-1-x)\cdot\omega_{x,j} + \sum_{y=1}^{x-1}\omega_{y,j}$ cannot be reduced by enlarging the maximum critical section of a (randomly chosen) task τ_i regarding resource ρ_j to η_j. The above process can be conducted iteratively to prove that the value of $(m-1-x)\cdot\omega_{x,j} + \sum_{y=1}^{x-1}\omega_{y,j}$ cannot be reduced by enlarging all x largest $|\rho_i^j|$ to η_j. In this case, $(m-1-x)\cdot\hat{\omega}_{x,j} + \sum_{y=1}^{x-1}\hat{\omega}_{y,j}$ can be represented as:

$$(m-1-x)\cdot\hat{\omega}_{x,j} + \sum_{y=1}^{x-1}\hat{\omega}_{y,j} = (m-1)\cdot x\cdot\eta_j - \frac{(x+1)^2-(x+1)}{2}\cdot\eta_j \tag{15}$$

Therefore, for any $0 < x \leq m-1$,

$$(m-1-x)\cdot\omega_{x,j} + \sum_{y=1}^{x-1}\omega_{y,j} \leq (m-1)\cdot x\cdot\eta_j - \frac{(x+1)^2-(x+1)}{2}\cdot\eta_j \tag{16}$$

which proves this lemma ■

Lemma 13. *During Λ_k, the contribution to the total interference that needs to be considered in our analysis is no more than:*

$$\Delta_k = \sum_{\rho_j\in\rho}(\psi_k^j \cdot \frac{m^2-3m+2}{2}\cdot\eta_j) \tag{17}$$

Proof: According to Lemma 12, during Λ_k, the contribution to the total interference that needs to be considered in our analysis is bounded by $(m-1)\cdot x\cdot\eta_j - \frac{(x+1)^2-(x+1)}{2}\cdot\eta_j$. It is trivial to prove that this function reaches its maximum value $\frac{m^2-3m+2}{2}$ when $x = m-1$. Therefore, we prove this lemma. ■

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 3, 4, 6, 11 and 13 gives an upper bound on the total interference to the problem job (a job of task τ_k) during its problem window $[a_k, a_k + D_k)$:

$$I_k \leq m\cdot B_{k,k} + \Upsilon_k + \Pi_k + \Delta_k + \Phi_k \tag{18}$$

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 $\tau_k \in \tau$,*

$$m\cdot B_{k,k} + \Upsilon_k + \Pi_k + \Delta_k + \Phi_k \leq m(D_k - C_k) \tag{19}$$

Proof: Follows from Theorem 2 and Equation (18). ■

VI. Analysis m-CDW

Initial experiments have been conducted to study the effectiveness of the *lp-CDW* analysis under different circumstances. Comparisons are made between the *lp-CDW* analysis and a variant of the *BL* analysis extended according to existing approaches introduced in section IV-A (named *WIA* in this report) to understand their different characteristics and justify the usefulness of the new analysis. Although the *lp-CDW* analysis performs better in many cases, it can sometimes be outperformed significantly, in terms of the number of recognised schedulable tasksets, by the *WIA* analysis. However, if reconfigured to study the schedulability of each individual task, our experiments reveal that the two analyses actually complement each other as they work better at different priorities. Consequently, we combine them (referred to as *m-CDW* ("m" stands for "mixed")) to apply different schedulability tests (*lp-CDW* or *WIA*) to different task priorities. The performance of this hybrid analysis is compared to the others' in section VII.

In this section, only a few experiments are presented to explain how *lp-CDW* and *WIA* complement each other. All these experiments simulate a four processor system in which only one resource is shared among tasks. Hence, there is not any nested resource access. 20000 tasksets are randomly generated for each total utilisation (x axis value) in every experiment. The success rates (the number of recognised

schedulable tasksets divided by 20000) of *WIA*, *lp-CDW* and *BL* (which does not consider any resource sharing) performed on the same 20000 tasksets are reported (y axis). Because the *BL* analysis does not consider any resource sharing and all other analyses are derived from it, the *BL* analysis effectively dominates all other analyses discussed in this report. Therefore, we use the performance of *BL* as a reference for the evaluation of other analyses. Details of experiments and more results are given in section VII.

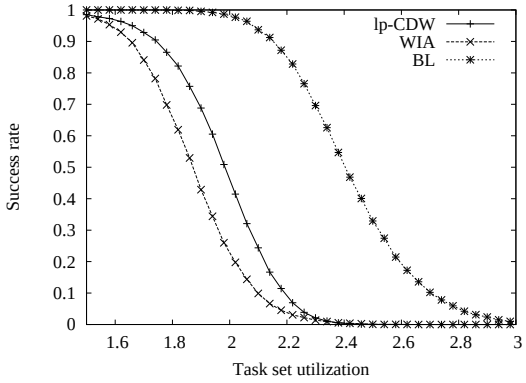

Figure 8. *lp-CDW* vs *WIA* when task number is 15 and every job accesses resource at most 5 times

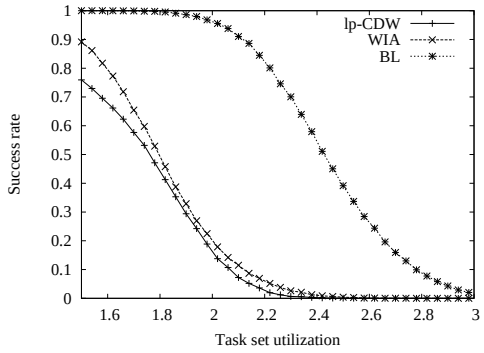

Figure 9. *lp-CDW* vs *WIA* when task number is 10 and every job accesses resource at most 10 times

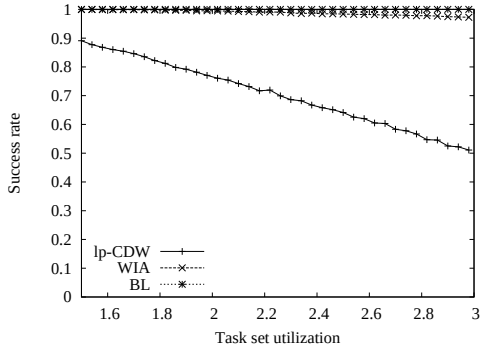

Figure 10. *lp-CDW* vs *WIA* at the highest priority only

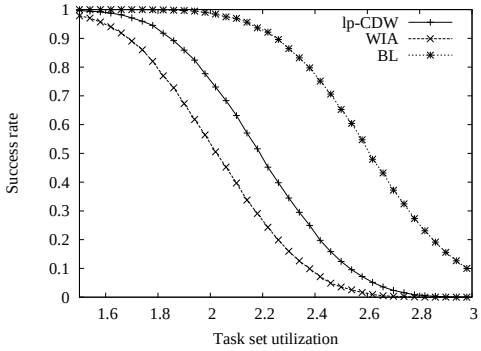

Figure 11. *lp-CDW* vs *WIA* at the lowest priority only

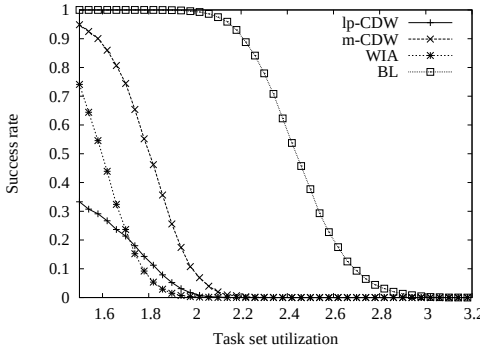

Figure 12. *m-CDW* vs *lp-CDW* vs *WIA* on 4 processors under DkC when task number is 20

In figure 8, each taskset consists of 15 tasks and *lp-CDW* outperforms *WIA* when the total utilisation of every taskset is set to between 1.5 and 2.3. When tested with other utilisations, *lp-CDW* and *WIA* give very similar results. However, as illustrated by figure 9, the performance of *lp-CDW* degrades when the maximum number of resource accesses in any job is changed from 5 (as in figure 8) to 10 and each taskset is modified to contain 10 tasks. With this configuration, *WIA* outperforms *lp-CDW* at all utilisations below 2.4.

In order to better understand the performance degradation, we break down the experiment given in figure 9 to show the schedulability analysis success rate of each individual task τ_i (the number of priority i tasks that are recognised as schedulable in all tasksets divided by 20000).

In figure 10, *lp-CDW* can recognise as few as just a little more than 50% of all the highest priority tasks as schedulable ones. However, *WIA* considers nearly all highest priority tasks schedulable. On the other hand, figure 11 shows the success rates of the lowest priority task according to the two analyses. In this figure, *lp-CDW* outperforms *WIA* significantly when the total utilisation of every taskset is set to between 1.5 and 2.7.

This trend is mirrored by experiments with tasks at other

priorities: the higher the priority, the better *WIA* performs; the lower the priority, the better *lp-CDW* performs. This is mainly caused by the different approaches used by *WIA* and *lp-CDW* to modeling spinning time. As explained previously, the *WIA* analysis assumes that every resource request always causes $(\hat{n}_j - 1) \cdot \eta_j$ spinning time. However, when analyzing task τ_k, the *WIA* analysis only considers tasks with priorities higher than τ_k. Consequently, only spinning executed by higher priority tasks will be counted. On the other hand, the *lp-CDW* analysis has to consider all tasks' resource accesses when estimating the total spinning time for the analysis of τ_k. The advantage of this analysis is that the estimation of the total spinning time of all the tasks within the problem window is much more accurate than *WIA*. Hence, when τ_k's priority is high, the pessimism of *WIA* can be compensated by ignoring lower priority tasks. When τ_k's priority is low, the pessimism of *WIA* increases significantly as it has to consider more and more tasks. However, as *lp-CDW* always considers all tasks' resource accesses when estimating total spinning time, it performs better than *WIA* when τ_k is at a low priority.

m-CDW combines both *WIA* and *lp-CDW*. It analyzes the schedulability of tasks one by one in the descending order of their priorities. At each priority, it first uses the *WIA* analysis and only if that fails is the *lp-CDW* analysis used. In the end, the *m-CDW* analysis fails if *lp-CDW* fails at any priority. Accordingly, the *m-CDW* analysis dominates both *WIA* and *lp-CDW*. As will be seen in section VII, in many experiments, *m-CDW* significantly outperforms both analyses of which it is composed (An example is given in figure 12.).

VII. Evaluations

In this section, we compare the performance of an existing WCET inflation analysis *WIA* (as given in section VI), Bertogna and Lipari's *BL* analysis [10] (which does not consider resource sharing) and both analyses proposed in this report (i.e. *lp-CDW* and *m-CDW*). First of all, we present the details of the experiment setup. Then, we compare all the above analyses empirically.

A. Methodology

In order to observe the behaviour of the proposed analyses in different circumstances, our experiments were conducted on randomly generated tasksets with different parameters specified. 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 of the randomly generated longest critical section of every resource accessing task.

All experiments in this report assume that only one resource exists. Therefore, no nested resource access can happen. The maximum 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}.

As can be seen shortly, the priority assignment policy has an impact on the schedulability of tasksets in many cases. The policies we tested include deadline monotonic *DM* (the longer the deadline, the lower the priority), $(D-C)$ monotonic *DCM* (the higher the $(D_i - C_i)$, the lower the priority) and $(D - k \cdot C)$ monotonic *DkC* (the higher the $(D_i - k \cdot C_i)$, the lower the priority; k is an coefficient calculated according to [5]).

Moreover, the total utilization of our tasksets ranges between 0 and m. The number of tasks in each taskset is varied between 10 and 25. 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* [26]. 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 pass each analysis. It should be noticed that we have no way of knowing how many tasksets are indeed schedulable. The figures from our experiments only show the number of schedulable tasksets that can be identified by our analyses. As explained in section VI, we use the performance of *BL* as a reference for the evaluation of other analyses.

B. Priority Assignment Policy

The goal of this experiment is not to find the optimal priority assignment policy for the proposed analyses. Instead, we intend to find out if any of the three concerned analysis can benefit more from a particular priority assignment policy. Also, we are interested in finding out if there is one priority assignment policy under which all analyses perform better than under other policies.

First, a $m = 4$ processor system is evaluated. The number of tasks in each taskset is set to 20 (Three other taskset sizes

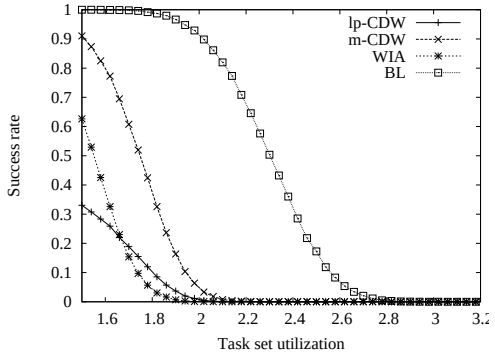

Figure 13. *m-CDW* vs *lp-CDW* vs *WIA* on 4 processors under DM

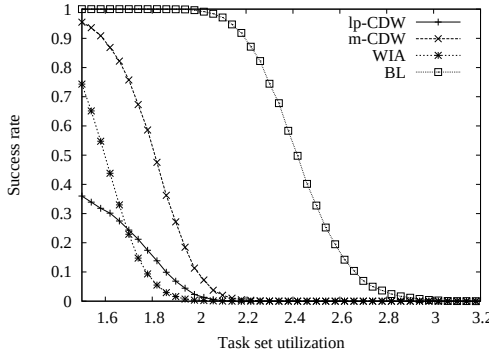

Figure 14. *m-CDW* vs *lp-CDW* vs *WIA* on 4 processors under DCM

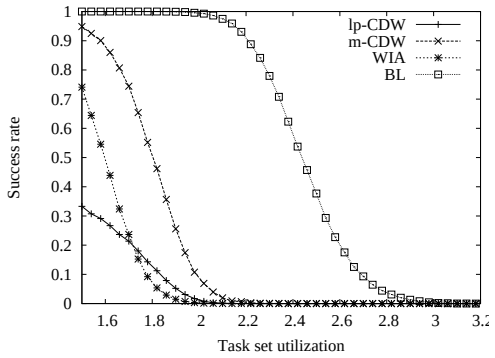

Figure 15. *m-CDW* vs *lp-CDW* vs *WIA* on 4 processors under DkC

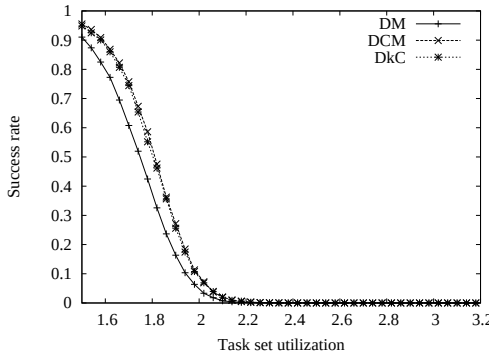

Figure 16. *m-CDW* on 4 processors under different priority assignment policies

10, 15 and 25 are also tested and the results are given in appendix B.). ψ^{bound}, CS^{lb} and CS^{ub} are always set to 5, 10 and 25 respectively.

Figures 13 to 15 depict the performance of all three concerned analyses under *DM*, *DCM* and *DkC* policies when each taskset consists of 20 tasks. The x axis in each figure denotes the total utilisation of each taskset. The y axis in each figure shows the success rate of each analysis (which is defined as the number of schedulable tasksets recognised by an analysis divided by the total number of tested tasksets, i.e. 20000 in our experiments.). As can be seen in these figures, the performance of *lp-CDW* (as a standalone analysis) is not as good as that of *WIA*. This is because *lp-CDW* overestimates the total spinning time in each high priority task's problem window. By contrast, *m-CDW* outperforms both *WIA* and *lp-CDW* substantially. This will be explained by other experiments shortly.

According to figures 13 to 15 (and those in appendix B), priority assignment policy does not have a big impact on the relative performance of the three concerned analyses. However, these figures do not clearly show whether any priority assignment policy is better than the others for the absolute performance of the concerned analyses. As the relative performance of each analysis has been illustrated in figures 13 to 15, only the *m-CDW* analysis is tested for the impact of priority assignment policies on its absolute performance. As illustrated by figure 16, the priority assignment policy *DkC* generally has a better performance than the other two policies. However, the performance difference between *DkC* and *DCM* is insignificant.

Next, we change the number of processors to $m = 8$ and keep all other parameters unmodified. As illustrated by figures 17 to 19, *m-CDW* still outperforms *WIA* significantly. The performance gap between *m-CDW* and *WIA* becomes even larger than the 4 processor case. It is also clear the *DM* policy is not as good as the other two policies in terms of the absolute performance of all three analyses.

The relative performance of *WIA* is worse in this case than in the 4 processor case because: 1) the increase of processor number has a direct impact on the spinning time of every resource access in *WIA* (The spinning time more than doubles); 2) *lp-CDW* and *m-CDW* are less influenced since it is more difficult to find many groups of x potentially parallel resource accesses when x is large.

As illustrated by figure 20, the priority assignment policy *DkC* always performs better than the other two policies for *m-CDW* in the 8 processor case. The performance of *DM* is particularly poor.

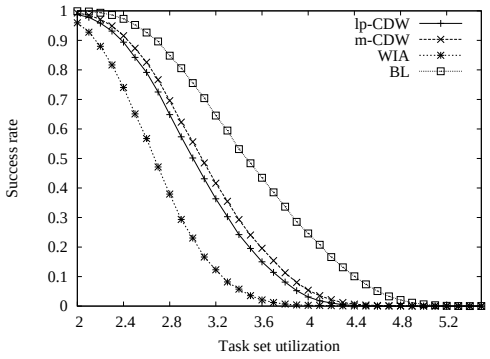

Figure 17. *m-CDW* vs *lp-CDW* vs *WIA* on 8 processors under DM

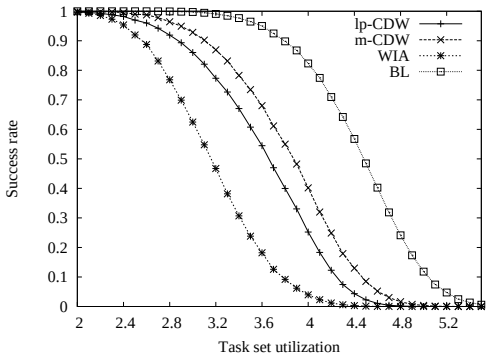

Figure 18. *m-CDW* vs *lp-CDW* vs *WIA* on 8 processors under DCM

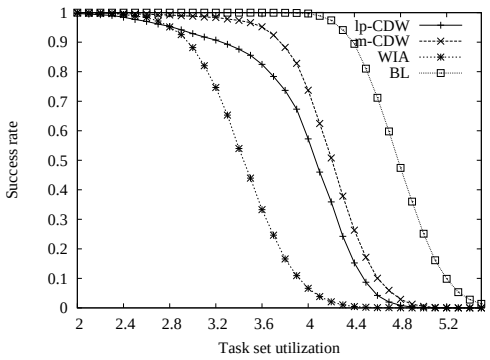

Figure 19. *m-CDW* vs *lp-CDW* vs *WIA* on 8 processors under DkC

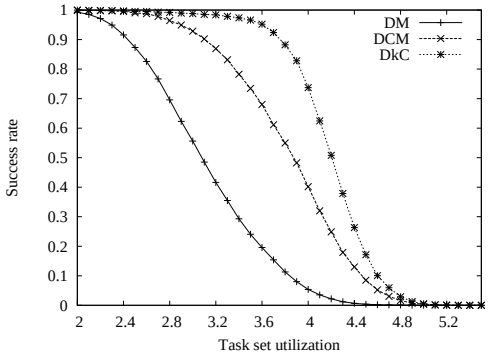

Figure 20. *m-CDW* on 8 processors under different priority assignment policies

C. Taskset Size

In this 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 and their priorities are assigned according to *DkC*. The number of tasks in each taskset is varied between 10 and 25. The total utilisation of each taskset tested in this experiment is chosen among 1.5, 2 and 2.5. ψ^{bound}, CS^{lb} and CS^{ub} are always set to 5, 5 and 20 respectively.

Figure 21 illustrates the performance of all three analyses with different taskset sizes when each taskset has a total utilisation of 1.5. The x axis in the following figures denotes the number of tasks in each taskset. The y axis shows the success rate of each analysis. Since the total utilisation of this test is relatively low, most tasksets are recognised as schedulable by *BL* and *m-CDW*. However, both the *WIA* and *lp-CDW* analyses begin to miss significant number of schedulable tasksets after task number reaches 16 under this configuration. The performance of *lp-CDW* drops even more quickly than *WIA* as taskset size increases.

After the total utilisation of each taskset is increased to 2, our test clearly shows (in figure 22) the performance of all three analyses degrades as task number increases. The performance of all three analyses degrades at similar speeds but the absolute performance of *m-CDW* is always better. It should also be noticed that the *lp-CDW* analysis becomes better than the *WIA* analysis at all task numbers in this more demanding test.

When the total utilisation of each taskset is set to 2.5 (figure 23), it becomes very hard for all three analyses to recognise any schedulable taskset. They can only recognise a few when the taskset size is low. Nevertheless, *m-CDW* still outperforms all other analyses.

Next, we change the number of processors to $m = 8$ and adjust the total utilisations of tasksets to 3, 3.5, 4 and 4.5 in different tests. The taskset sizes are still varied between 10 and 25.

At utilisation 3 (figure 24), *m-CDW* can recognise nearly all tasksets schedulable irrespective of the number of tasks in each taskset. Although *lp-CDW* cannot recognise as many schedulable tasksets as *m-CDW*, its performance is also hardly affected by the increase of taskset size. However, the success rate of *WIA* drops as the taskset size increases.

When total utilisation is raised to 3.5 (25), *m-CDW* can still recognise nearly all tasksets schedulable at all task numbers. The success rate of *lp-CDW* becomes lower than the previous test but the impact of taskset size changes on its performance is still limited. The performance of *WIA* degrades more quickly than the other two analyses along the x axis. As illustrated by figures 26 and 27, the performance of *WIA* degrades more and more quickly along the x axis

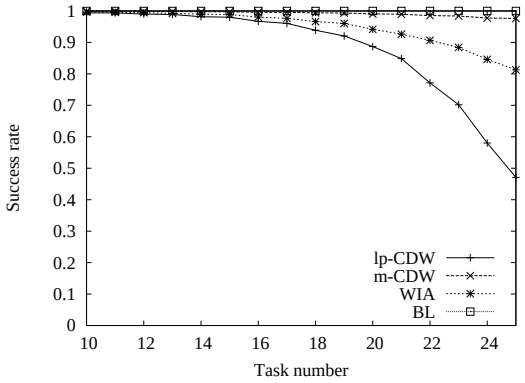

Figure 21. *m-CDW* vs *lp-CDW* vs *WIA* on 4 processors when the total utilisation is 1.5 and task number varies between 10 and 25

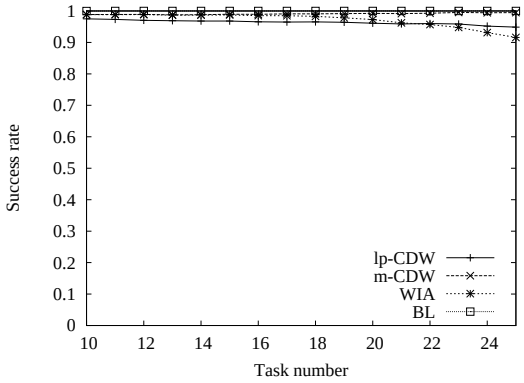

Figure 24. *m-CDW* vs *lp-CDW* vs *WIA* on 8 processors when the total utilisation is 3 and task number varies between 10 and 25

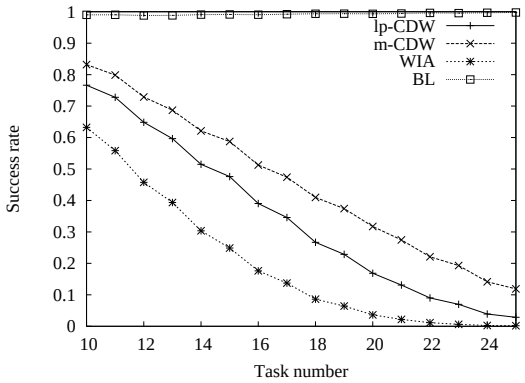

Figure 22. *m-CDW* vs *lp-CDW* vs *WIA* on 4 processors when the total utilisation is 2.0 and task number varies between 10 and 25

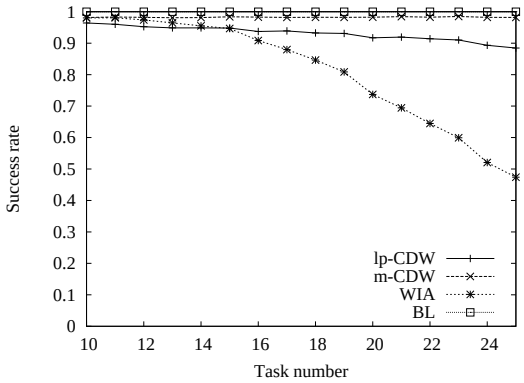

Figure 25. *m-CDW* vs *lp-CDW* vs *WIA* on 8 processors when the total utilisation is 3.5 and task number varies between 10 and 25

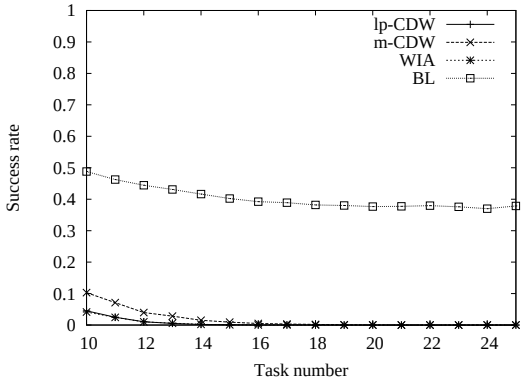

Figure 23. *m-CDW* vs *lp-CDW* vs *WIA* on 4 processors when the total utilisation is 2.5 and task number varies between 10 and 25

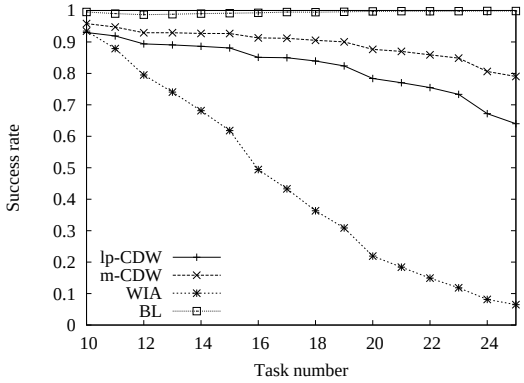

Figure 26. *m-CDW* vs *lp-CDW* vs *WIA* on 8 processors when the total utilisation is 4.0 and task number varies between 10 and 25

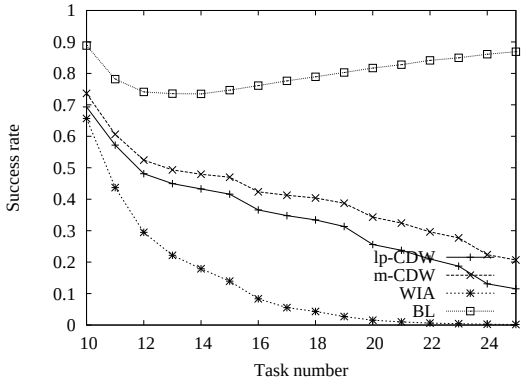

Figure 27. *m-CDW* vs *lp-CDW* vs *WIA* on 8 processors when the total utilisation is 4.5 and task number varies between 10 and 25

as the utilisation further increases. On the other hand, *m-CDW* and *lp-CDW* are more resilient to both the increase of taskset utilisation and the increase of taskset size.

D. ψ^{bound} — maximum number of resource accesses

This experiment studies the behaviour of all three analyses when the maximum number of resource accesses per job ψ^{bound} is set differently. Suppose all tasks are ordered according to *DkC* and execute on $m = 4$ processors. Each taskset consists of 10 tasks. CS^{lb} and CS^{ub} are set to 5 and 15 respectively. In each test, ψ^{bound} is set to one of the following values $(5, 10, 15, 20)$.

As in figure 28 where ψ^{bound} is set to 5, the success rates of all three analyses are relatively close. Moreover, the performance of all three analyses is also relatively close to *BL* in this test. This is because the low ψ^{bound} set for this test causes not only smaller number of resource accesses in individual tasks but also a lower total resource access number (according to the methodology of our evaluation). As ψ^{bound} increases (figures 29 to 31), the performance gap between *m-CDW* and *BL* grows larger, so does the performance gap between *WIA* and *m-CDW*. Although *lp-CDW* plays an important role in the performance improvement of *m-CDW* over *WIA*, this analysis, when used alone, is much less resilient to the ψ^{bound} increase.

Next, in order to see more clearly the impact of ψ^{bound} on the absolute performance of each analysis, we vary ψ^{bound} between 5 and 20 at the step of 1 for three particular total utilisations: 1.5, 2 and 2.5. At utilisation 1.5 (figure 32), the success rate of *m-CDW* can drop to as low as around 80% when $\psi^{bound} = 20$. The success rate of *WIA* drops more quickly than that of *m-CDW* at high ψ^{bound} values. The quick performance degradation of *lp-CDW* along the x axis echoes the previous tests. At higher utilisations (figures 33 and 34), the performance of *m-CDW* remains better than the other two analyses. The success rates of all three analyses

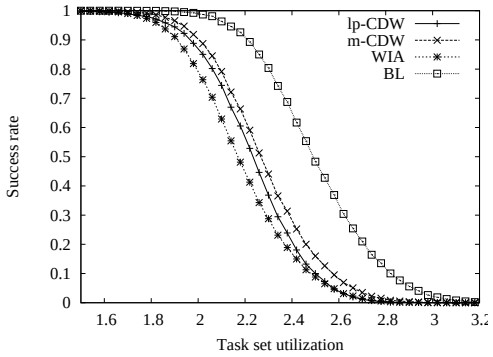

Figure 28. *m-CDW* vs *lp-CDW* vs *WIA* when $\psi^{bound} = 5$

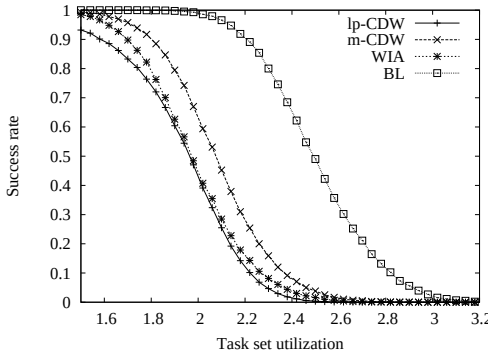

Figure 29. *m-CDW* vs *lp-CDW* vs *WIA* when $\psi^{bound} = 10$

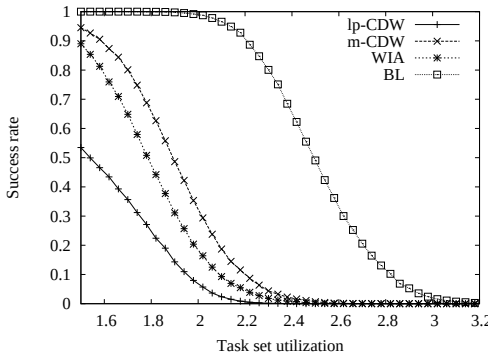

Figure 30. *m-CDW* vs *lp-CDW* vs *WIA* when $\psi^{bound} = 15$

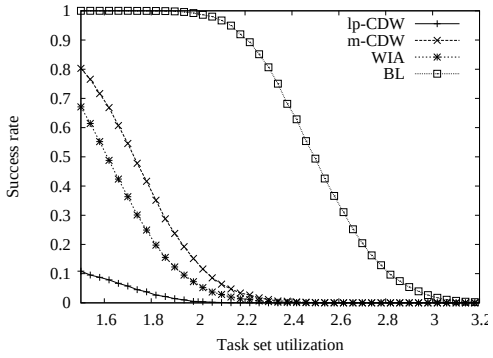

Figure 31. *m-CDW* vs *lp-CDW* vs *WIA* when $\psi^{bound} = 20$

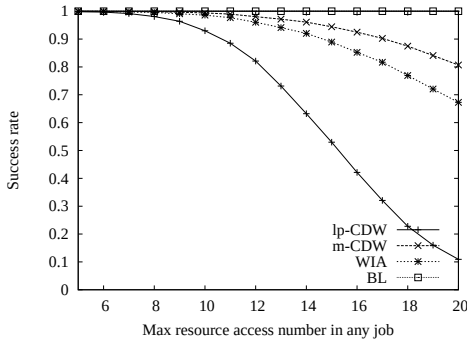

Figure 32. *m-CDW* vs *lp-CDW* vs *WIA* with different ψ^{bound} when the total utilisation is 1.5

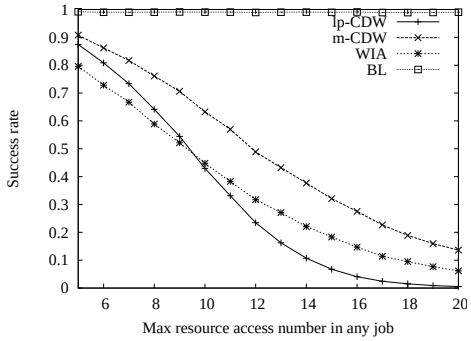

Figure 33. *m-CDW* vs *lp-CDW* vs *WIA* with different ψ^{bound} when the total utilisation is 2.0

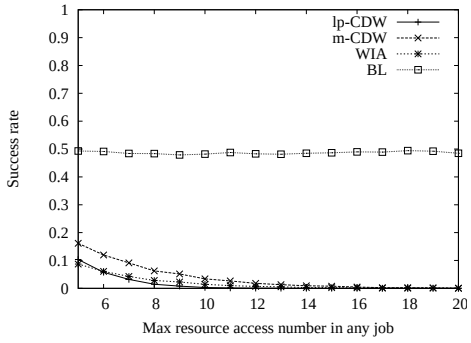

Figure 34. *m-CDW* vs *lp-CDW* vs *WIA* with different ψ^{bound} when the total utilisation is 2.5

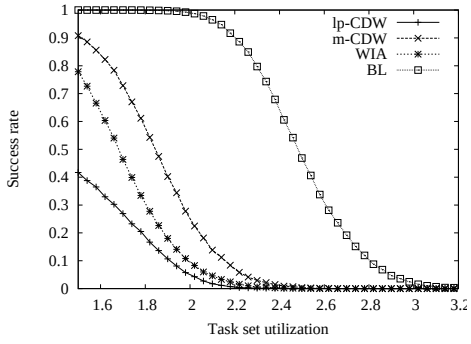

Figure 35. *m-CDW* vs *lp-CDW* vs *WIA* when $CS^{lb} = 5$

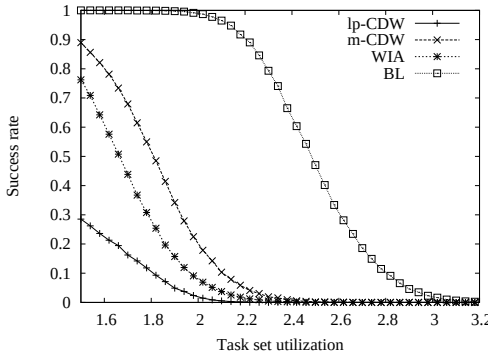

Figure 36. *m-CDW* vs *lp-CDW* vs *WIA* when $CS^{lb} = 10$

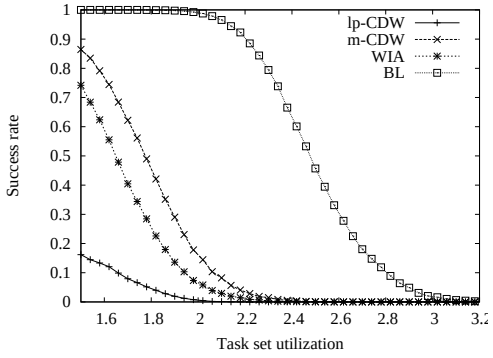

Figure 37. *m-CDW* vs *lp-CDW* vs *WIA* when $CS^{lb} = 15$

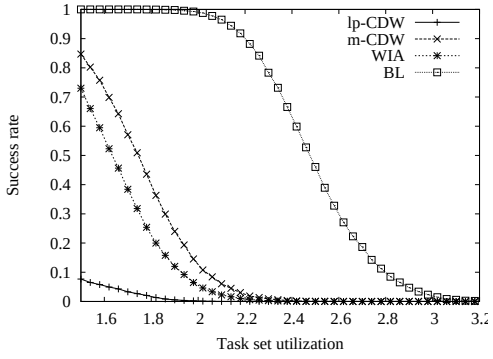

Figure 38. *m-CDW* vs *lp-CDW* vs *WIA* when $CS^{lb} = 20$

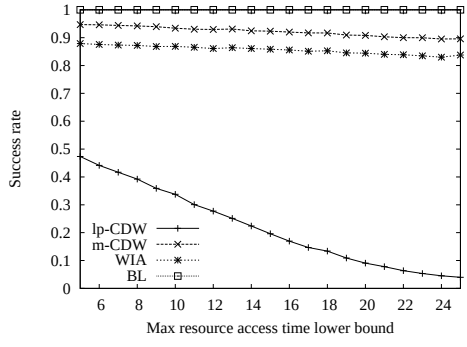

Figure 39. *m-CDW* vs *lp-CDW* vs *WIA* with different CS^{lb} when the total utilisation is 1.4

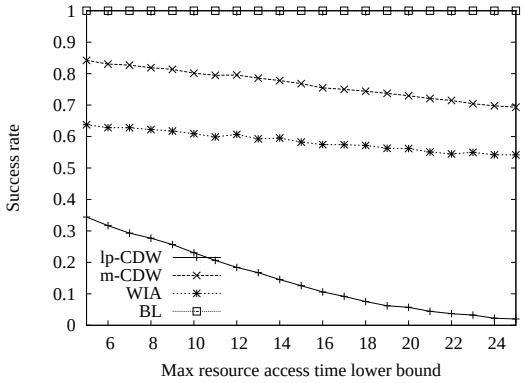

Figure 40. *m-CDW* vs *lp-CDW* vs *WIA* with different CS^{lb} when the total utilisation is 1.6

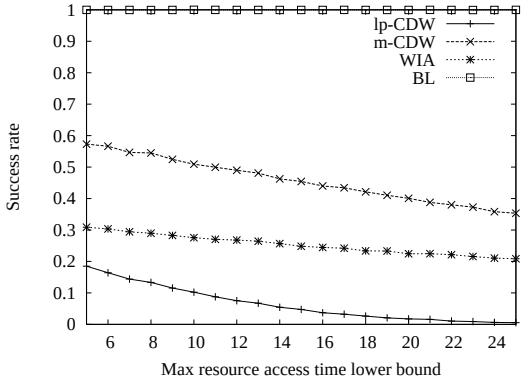

Figure 41. *m-CDW* vs *lp-CDW* vs *WIA* with different CS^{lb} when the total utilisation is 1.8

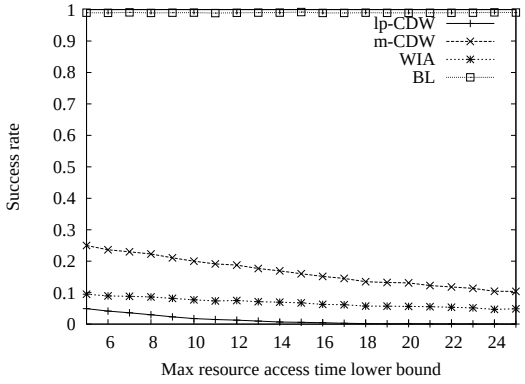

Figure 42. *m-CDW* vs *lp-CDW* vs *WIA* with different CS^{lb} when the total utilisation is 2.0

drop monotonically as the maximum number of resource accesses per job increases.

E. CS^{lb} — lower bound of longest critical section

Next, we investigate how different CS^{lb} values affect the performance of our analyses when CS^{ub} is constant. Again, all tasks execute on $m = 4$ processors and are ordered according to *DkC*. Each taskset is composed of 10 tasks. ψ^{bound} and CS^{ub} are set to 10 and 25 respectively. In each test, CS^{lb} is set to one of the following values $(5, 10, 15, 20)$.

According to figures 35 to 38, the change of CS^{lb} has very little effect on the relative performance of our analyses. In order to see more clearly the impact of this change on the absolute performance of each analysis, CS^{lb} is varied between 5 and 25 at the step of 1 for 4 different utilisations: 1.4, 1.6, 1.8 and 2. As can be seen in figures 39 to 42, *WIA* is the most resilient to the CS^{lb} variation though its performance is not as good as that of *m-CDW*. This can be explained by examining equation 1 in section VI. In this equation, *WIA* assumes every request for resource ρ_j wastes $\eta_j \cdot (\hat{n}_j - 1)$ time units on spinning. As $\eta_j = \max_i(|\rho_i^j|)$, CS^{lb} has very little influence on it as long as CS^{ub} remains unchanged. Hence, *WIA* is less sensitive to the change of CS^{lb} than the other two analyses, which both consider the $|\rho_i^j|$ differences.

Moreover, the performance of *lp-CDW* degrades more quickly than that of *m-CDW* as CS^{lb} increases.

F. CS^{ub} — upper bound of longest critical section

By contrast, in this experiment, we investigate how different CS^{ub} values affect the performance of our analyses when CS^{lb} is constant. Again, all tasks execute on $m = 4$ processors and are ordered according to *DkC*. Each taskset is composed of 10 tasks. ψ^{bound} and CS^{lb} are set to 10 and 5 respectively. In each test, CS^{ub} is set to one of the following values $(15, 20, 25, 30)$.

According to figures 43 to 46, the performance gap between *m-CDW* and *BL* becomes larger and larger as CS^{ub} increases, so does the gap between *WIA* and *m-CDW*. The *lp-CDW* analysis is much less resilient to the CS^{ub} variation than the other two analyses.

Next, we vary CS^{ub} between 15 and 30 at the step of 1 for 4 different utilisations: 1.4, 1.6, 1.8 and 2. As can be seen in figures 47 to 50, *m-CDW* is the most resilient to the CS^{ub} variation as it degrades slower along the x axis. In contrast to the CS^{lb} experiments, the performance of *WIA* degrades quickly as CS^{ub} increases. This is because CS^{ub} has a greater impact on η_j.

The success rate of *m-CDW* is always better than the other two analyses.

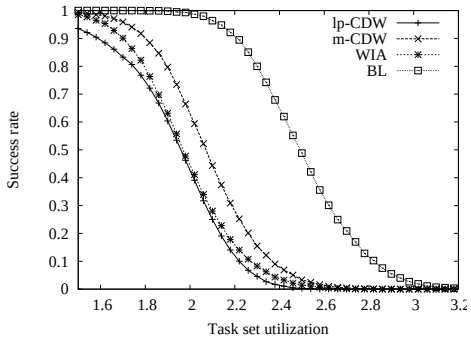

Figure 43. *m-CDW* vs *lp-CDW* vs *WIA* when $CS^{ub} = 15$

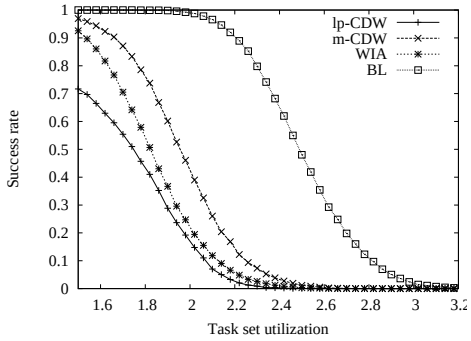

Figure 44. *m-CDW* vs *lp-CDW* vs *WIA* when $CS^{ub} = 20$

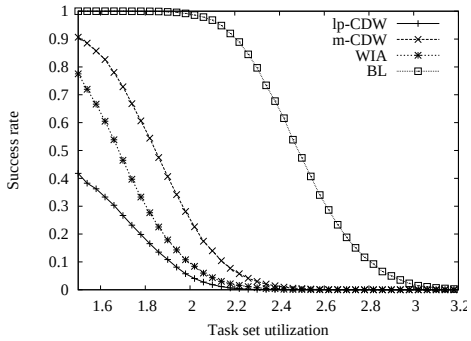

Figure 45. *m-CDW* vs *lp-CDW* vs *WIA* when $CS^{ub} = 25$

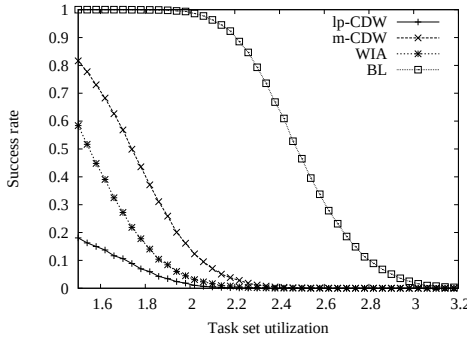

Figure 46. *m-CDW* vs *lp-CDW* vs *WIA* when $CS^{ub} = 30$

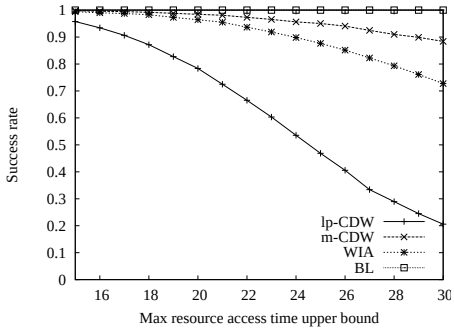

Figure 47. *m-CDW* vs *lp-CDW* vs *WIA* with different CS^{ub} when the total utilisation is 1.4

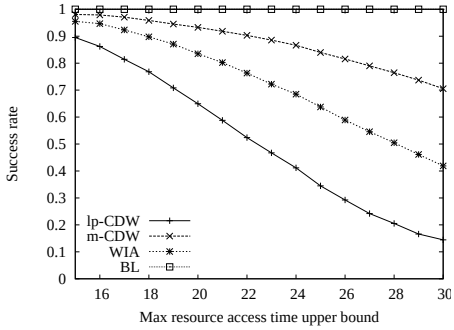

Figure 48. *m-CDW* vs *lp-CDW* vs *WIA* with different CS^{ub} when the total utilisation is 1.6

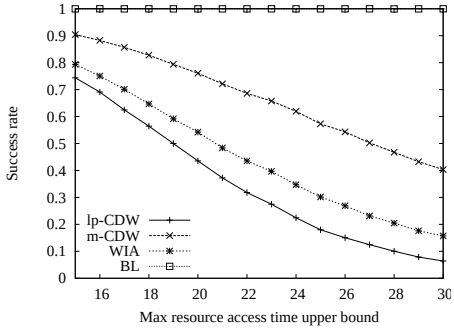

Figure 49. *m-CDW* vs *lp-CDW* vs *WIA* with different CS^{ub} when the total utilisation is 1.8

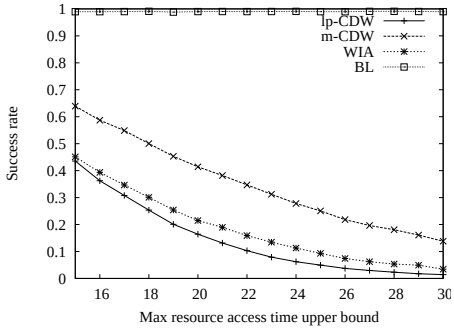

Figure 50. *m-CDW* vs *lp-CDW* vs *WIA* with different CS^{ub} when the total utilisation is 2.0

G. m-CDW *Internal*

As explained in section VI, *WIA* and *lp-CDW* work better for high priority and low priority tasks respectively. Therefore, the *m-CDW* analysis applies *WIA* analysis to every task in the descending order of their priorities until the first time the *WIA* analysis fails. Then *lp-CDW* will be more often used to analyze all the remaining tasks and the task deemed to be unschedulable by *WIA*. For ease of presentation, we name the priority at which a *m-CDW* analysis begins to use *lp-CDW*, the *critical priority*.

It should be noticed that different tasksets may have different critical priorities though they are all analyzed according to *m-CDW*. A taskset may have no critical priority if the whole taskset is schedulable according to *WIA*.

In this experiment, we study different tasksets to find out which priorities are more likely to become critical priorities. This helps us better understand how the use of *lp-CDW* improves *WIA* so dramatically. All tasks execute on $m = 4$ processors and are ordered according to *DkC*. Each taskset consists of 15 tasks. CS^{lb} and CS^{ub} are set to 5 and 20 respectively. In each test, ψ^{bound} is set to one of the following values $(5, 10, 15)$. The total utilisation of each taskset is varied between 1.5 and 2.5 at the step of 0.2.

For each combination of total utilisation and ψ^{bound}, we randomly generate 20000 tasksets in the same way as all previous experiments. Then this experiment records the total number of tasksets that are considered schedulable by *m-CDW* (S_{total}). For each priority, this experiment also records the total number of times it becomes a critical priority in any recognised schedulable taskset (P_i, where i denotes a priority). Finally, this experiment reports each priority's possibility of becoming a critical priority in the form of $\frac{P_i}{S_{total}}$ (y axis).

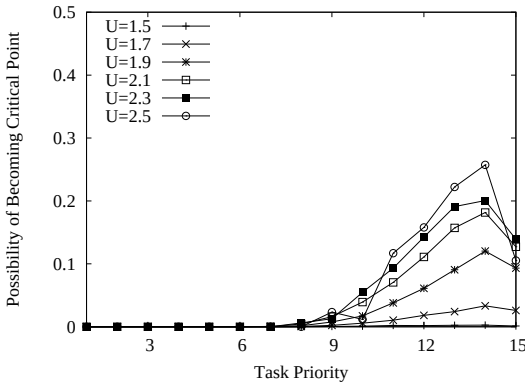

Figure 51. Possibility of each priority to become a critical priority when $\psi^{bound} = 5$

In figure 51, there are at most 5 resource accesses in any job of any taskset. When the total utilisation is 1.5, no priority shows a high possibility of becoming a critical priority, which means most schedulable tasksets are purely recognised by *WIA*. As utilisation increases, each priority's possibility of becoming a critical priority becomes higher and higher. When utilisation is 2.5, more than 25% of all the recognised schedulable tasksets have a critical priority of 14. Also in this case, more than 10% recognised schedulable tasksets need only use *lp-CDW* at the lowest priority. However, *m-CDW* is so critical that around 86% of all recognised schedulable tasksets have their own critical priorities, which means they cannot be found schedulable by *WIA* alone. Also in this figure, priority as high as 7 (1 represents the highest priority) can become a critical priority (when utilisation is 1.9 or 2.1).

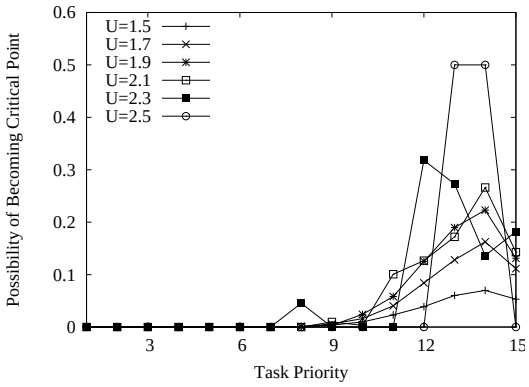

Figure 52. Possibility of each priority to become a critical priority when $\psi^{bound} = 10$

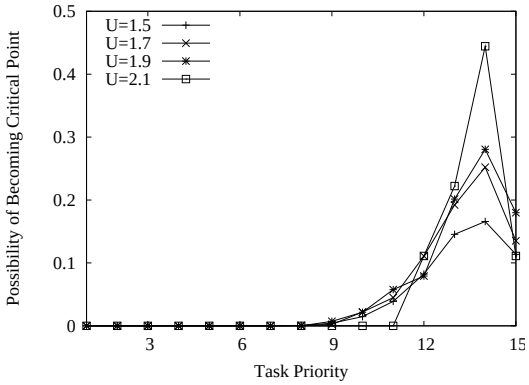

Figure 53. Possibility of each priority to become a critical priority when $\psi^{bound} = 15$

According to figure 52, when utilisation is 2.5 and $\psi^{bound} = 10$, half of the recognised schedulable tasksets have a critical priority of 13 and the other half have a critical priority of 14. In this case, all schedulable tasksets have their own critical priorities (either 13 or 14). As illustrate by figure 53, increasing ψ^{bound} to 15 makes some priority's possibility of becoming a critical priority even higher or in some other cases, forces higher priorities to become critical priorities. The $U = 2.3$ and $U = 2.5$ cases are ignored

in this figure because even *m-CDW* cannot recognise any schedulable taskset at these utilisations.

Our experiments showed that *m-CDW* has significant better performance than *WIA* and the extent of this improvement tends to increase as the number of processors, taskset sizes and the upper bound on the critical section length increase.

VIII. Conclusions and Future Work

In this report, we provided 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 (global fixed task priority preemptive scheduling) multiprocessor scheduling that requires the use of queue locks (FIFO-queue-based non-preemptive spin 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 [16].

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 report how this approach introduces a significant amount of pessimism, especially at low priorities. This motivated the development of a new schedulability analysis, which takes a unique way to model spinning. Instead of inflating the worst-case execution time of every task, the new analysis *lp-CDW* groups potentially parallel resource requests (to ignore those that can never be in parallel with others) and considers tasks' differences in their maximum critical section lengths when modeling the total spinning time.

Experiments showed that although *lp-CDW* outperforms the existing approach (*WIA* as an example) in many cases, there are situations where *WIA* gives much better results. A close investigation on the failure rates at different priorities revealed the true characteristic of *lp-CDW* and *WIA* — they complement each other. When analyzing the schedulability of high priority tasks, *WIA* usually shows a much higher success rate. However, the lower the task priority is, the poorer the performance of *WIA* is and the better the performance of *lp-CDW* is. Consequently, we combined both *WIA* and *lp-CDW* to create a better hybrid analysis called *m-CDW*. Our studies showed that it is usually only necessary to use the *lp-CDW* part of *m-CDW* for tasks at the lowest priorities. However, some experiments also show that nearly no taskset can be identified schedulable without using the *lp-CDW* part of *m-CDW*.

The *m-CDW* analysis dominates *WIA* and *lp-CDW* and outperforms both of them significantly in most experiments conducted. Our experiments also suggest that the most effective priority ordering available for the proposed analyses is DkC monotonic. Taskset utilisation, the total number of tasks in each taskset, the number of processors, the maximum number of resource accesses allowed in each job and the lengths of resource accesses all have a big impact on the performance of the above analyses.

It remains an open question whether other spin lock algorithms can also benefit from the proposed analyses. Such algorithms may make use of prioritised queues, preemptive execution and/or priority inheritance [17], [18]. It is also unclear how to apply the intuition behind this work to multiprocessor systems based on 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 not allowed in the proposed analyses or assumed to share a common lock.

References

[1] R. Davis and A. Burns, "A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems," University of York, Department of Computer Science, Tech. Rep. YCS-2009-443, 2009.

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

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

[4] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah, "A categorization of real-time multiprocessor scheduling problems and algorithms," in *Handbook on Scheduling Algorithms, Methods, and Models*. Chapman Hall/CRC, Boca, 2004, http://www.cs.unc.edu/∼anderson/papers/multibook.pdf.

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

[6] B. Andersson, S. Baruah, and J. Jonsson, "Static-priority scheduling on multiprocessors," in *Proceedings of the 22nd IEEE Real-time Systems Synmposium*, 2001, pp. 193–202.

[7] M. Bertogna, M. Cirinei, and G. Lipari, "New schedulability tests for real-time task sets scheduled by deadline monotonic on multiprocessors," in *Proc. 9th International Conference on Principles of Distributed Systems*, 2005.

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

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

[10] 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.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] 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.

[17] T. Johnson and K. Harathi, "A prioritized multiprocessor spin lock," *IEEE Transactions on Parallel and Distributed Systems*, vol. 8, no. 9, pp. 926–933, 1997.

[18] C. Wang, H. Takada, and K. Sakamura, "Priority inheritance spin locks for multiprocessor real-time systems," in *Proceedings of ISPAN*, 1996.

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

[20] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *JACM*, vol. 20, no. 1, pp. 46–61, 1973.

[21] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. Wellings, "Applying new scheduling theory to static priority pre-emptive scheduling," *Software Engineering Journal*, vol. 8, no. 5, pp. 284–292, 1993.

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

[23] C. Phillips, C. Stein, E. Torng, and J. Wein, "Optimal time-critical scheduling via resource augmentation," in *Proceedings of the 29th Annual ACM Symposium on Theory of Computing*, 1007.

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

[25] S. Baruah and J. Goossens, "The EDF scheduling of sporadic task systems on uniform multiprocessors," in *Proceedings of the 29th Real-Time Systems Symposium, 2008*, 2008, pp. 367–374.

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

APPENDIX A.
PROOF OF THEOREM 1

In this appendix, we prove Theorem 1. First, we assume that all the requests in a given problem window have been grouped according to the algorithm described in the theorem (e.g. Algorithm (1)). For ease of presentation, we refer to this state as the *original state*. Two grouping results are considered *equivalent* if the numbers of groups of each size in both results are identical.

Our proof for this theorem works iteratively.

On each iteration, we first randomly choose a resource request group and reduce its size by one. Then, we randomly choose another group and increase its size by one as long as this increase of group size does not violate the requirement that resource requests in the same group must be issued by different tasks. If a group of size x is changed to size $x-1$, then size $x-1$ will be called a *negative dirty size*. On the other hand, if a group of size x is changed to size $x+1$, then size $x+1$ will be called a *positive dirty size* (see figure 54).

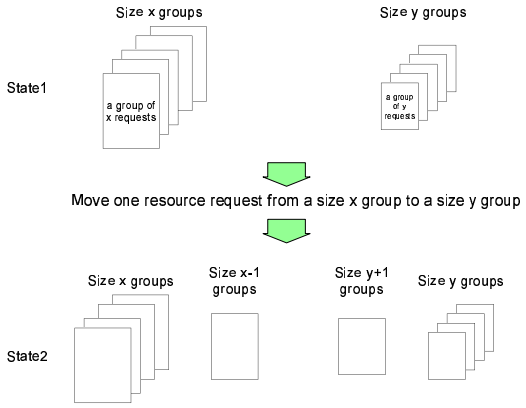

Figure 54. In state2, $x-1$ is a negative dirty size and $y+1$ is a positive dirty size

An iteration is a *necessary iteration* if it neither enlarges any group of a negative dirty size; nor reduces any group of a positive dirty size. Otherwise, it is an *unnecessary iteration*. For any sequence of iterations that contains some unnecessary iterations, a sequence of necessary iterations can always be found to generate an equivalent grouping result. Therefore, we can safely ignore all unnecessary iterations in the proof of this theorem. By repeating necessary iterations indefinitely, we can get all possible valid grouping results. Proving Theorem 1 is equivalent to proving that after

any number of necessary iterations, the new set of groups always cause no more total spinning time than the original state.

Definition 2. *An iteration is called a* safe iteration *if the group chosen to reduce size in this iteration was (before this iteration) larger than the group chosen to increase size. All other iterations are called* unsafe iterations.

For example, the iteration illustrated in figure 54 is a safe iteration as $x > y$.

Lemma 14. *For any sequence of necessary iterations starting from the original state, it contains only safe iterations.*

Proof:

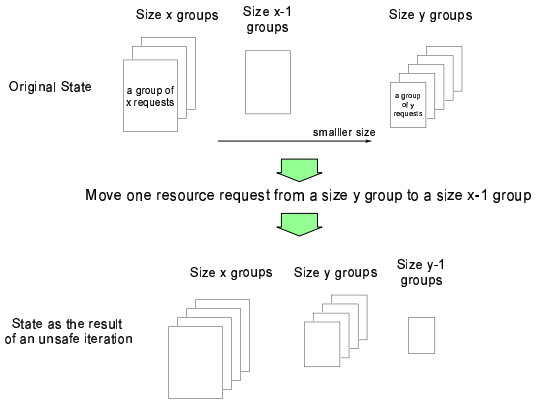

Figure 55. The first iteration cannot be an unsafe iteration

First, let's consider the first iteration which is performed when the groups are in the original state. In this case, any iteration is a necessary iteration as there is not any previous iteration. The number of groups of size x cannot be increased by reducing the number of groups of size y if $x - 1 \geq y$ (figure 55). This is because the result of such an operation would be one more size x group compared to the original state while all groups of sizes greater than x are identical to the original state (figure 55). This is impossible according to the algorithm described in Theorem 1. Consequently, the first iteration has to be a safe iteration.

Next, we choose an integer $i \geq 1$ and assume that all the i iterations made since the original state are all necessary safe iterations and the $(i+1)$th iteration is a necessary unsafe iteration. Then, we denote by $pdss_i$ ($ndss_i$), the set of all positive (negative) dirty sizes after i iterations. We also denote by $min(pdss_i)$ ($max(ndss_i)$) the minimum positive (maximum negative) dirty size. Because all of the first i iterations are necessary and safe, $max(ndss_i) \geq min(pdss_i)$.

There is only one way to make the $(i+1)$th iteration a necessary but unsafe iteration:

- choose a group size z where $z \notin pdss_i$; reduce a group from size z to size $z-1$ and transform a group of size $(q \geq z) \wedge (q \notin ndss_i)$ to $q+1$.

There are 4 cases to consider:

Case 1

If $q > max(ndss_i)$, the result of the $(i+1)$th iteration is one more size $(q+1)$ group compared to the original state while all groups of sizes greater than $(q+1)$ are identical to the original state (figure 56). This is impossible according to the algorithm described in Theorem 1.

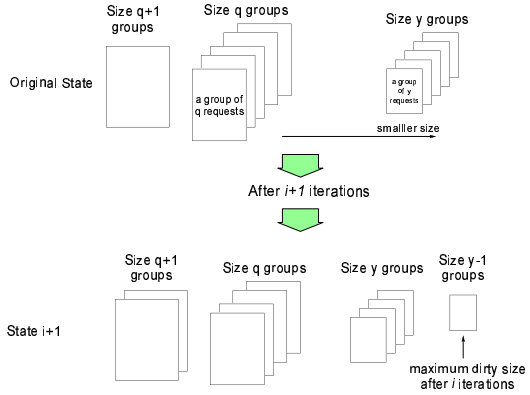

Figure 56. The $(i+1)$th iteration cannot be an unsafe iteration (situation1)

Case 2

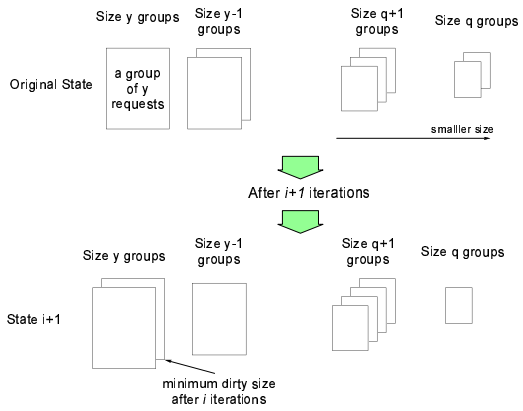

Figure 57. The $(i+1)$th iteration cannot be an unsafe iteration (situation2)

If, immediately before the $(i+1)$th iteration, $q < min(pdss_i)$, groups of sizes q and z have never been touched by the previous i iterations. Then, we undo the operations of all the previous i iterations in reverse order. Because operations of all the first i iterations are valid, each undo operation only moves a resource access from a group back to where is was in a previous valid state. Consequently, if the $(i+1)$th iteration is a valid one, the state must remain

valid after all the undo operations. Hence, if the result after all the undo operations is invalid, the $(i+1)$th iteration must also be an invalid one. In this case, the result of our undo operations is one more size $(q+1)$ group compared to the original state while all groups of sizes greater than $(q+1)$ are identical to the original state (figure 57). This is impossible according to the algorithm described in Theorem 1 and implies that the $(i+1)$th iteration in this case is invalid.

Case 3

If, immediately before the $(i+1)$th iteration, $max(ndss_i) > q \geq min(pdss_i)$ and $z < min(pdss_i)$ (figure 58), we undo, after the $(i+1)$th iteration, the operations of all the previous i iterations in reverse order. The groups of size $z-1$ have no effect on the undo operations because $z < min(pdss_i)$. As the $(i+1)$th iteration enlarges a group of size q, q can never be a negative dirty size, which implies that either groups of size q have never been touched by the first i iterations, or they were obtained by only adding resource accesses into them. This means that our undo operations either ignore groups of size q or remove resource accesses from these groups. Consequently, we can undo the first i iterations without undoing the result of the $(i+1)$th iteration, which enlarges a group of size q to $q+1$. The result of our undo operations is equivalent to moving one request from a group of size z to another group of size $q' \geq min(pdss_i)-1$ in the original state while all groups of sizes greater than $(q'+1)$ are identical to the original state. This is because, according to the definition of $min(pdss_i)$, no group that is smaller than $min(pdss_i)-1$ in the original state can ever be increased to size q after the i iterations. Therefore, after all the undo operations, a group of size $(q+1)$ can only be changed to some size of $q'+1 \geq min(pdss_i)$. This result is invalid according to the algorithm described in Theorem 1 because $q' \geq z$ ($q' \geq min(pdss_i)-1$ and $z < min(pdss_i)$).

Case 4

If, immediately before the $(i+1)$th iteration, $max(ndss_i) > q \geq z > min(pdss_i)$, we undo, after the $(i+1)$th iteration, the operations of all the previous i iterations. As the $(i+1)$th iteration reduces a group of size z, z can never be a positive dirty size, which implies that either groups of size z have never been touched by the first i iterations, or they were obtained by only removing resource accesses from them. Similarly, either groups of size q have never been touched by the first i iterations, or they were obtained by only adding resource accesses into them. This means that our undo operations either ignore groups of size q (z) or remove (add) resource accesses from (into) groups that have the size of q (z) immediately after the first i iterations. Consequently, we can undo the first i iterations without undoing the result of the $(i+1)$th iteration, which enlarges a group of size q to $q+1$ and reduces a group of size z to $z-1$. The result of these undo operations is equivalent to moving one request from a group of size $z' \geq z$ to another group of size $q' \geq min(pdss_i)-1$ in the original state while all other groups are identical to the original state. In this case, $q \geq z'$ because otherwise q would become a negative dirty size before the $(i+1)$th iteration (Since $z' > q \geq z$ and a group can only change its size by at most one in each iteration, a group of size z' has to become size q before it reaches a size of z.), which implies that the $(i+1)$th iteration would be an unnecessary operation.

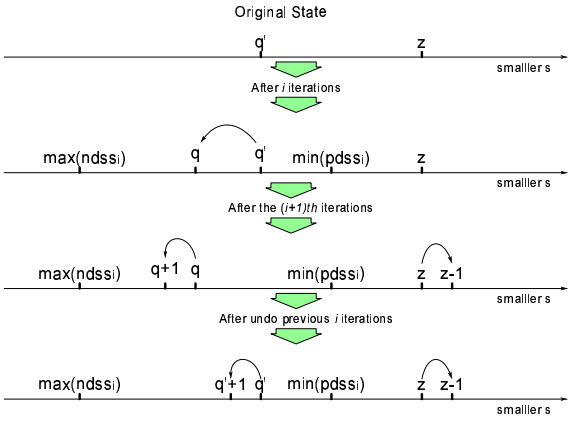

Figure 58. The $(i+1)$th iteration cannot be an unsafe iteration, situation 3 (An arrow from x to y means a group of size x is increased/decreased to size y)

If $q' \geq z'$, the result of the undo operations is invalid because the result contains one more size $q'+1$ group compared to the original state while all groups of sizes greater than $q'+1$ are identical to the original state. This is impossible according to the algorithm described in Theorem 1. In the cases where $z < q' < z'$, because $q \geq z'$, a negative and positive dirty point can always be formed during the first i iterations, which prevents both groups of sizes q' and z' from reaching sizes q and z. This makes the $(i+1)$th operation impossible. If otherwise, $q' \leq z < z'$, then z must be a positive dirty size after the first i iterations (Since $q' \leq z < z' \leq q$ and a group can only change its size by at most one in each iteration, a group of size q' has to become size z before it reaches a size of q.). This makes the $(i+1)$th iteration an unnecessary iteration, which contradicts our assumption. Consequently, in this case, the $(i+1)$th iteration cannot be an unsafe iteration, which completes the proof. ■

Theorem 1. *Suppose there is an algorithm that always makes as many size x parallel request groups as possible where x is initially set to $\hat{n}_j$ and decreases 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 $\sum_{x=\hat{n}_j}^{2}(x-1)\omega_{x,j}\cdot g_x$.

Proof: As a result of Lemma 14, all the *unsafe iterations* can be ignored when proving Theorem 1. Therefore, we only need to prove that after any number of necessary safe iterations, the new set of groups always cause no more total spinning time than the original state.

First, suppose $2 < x \leq m$. For a request group of size x, its maximum total spinning time is $(x-1)\omega_{x,j}$. If the size of this group is reduced to $x-1$, its maximum total spinning time will be reduced by $(x-1)\omega_{x,j} - (x-2)\omega_{x-1,j}$.

Because all iterations have to be both necessary and safe, the removed request cannot be added to a group of size $y \geq x-1$.

If the removed request is added to a group of size $y \leq x-2$, we will get a new group of size $y+1$. This introduces $(y+1-1)\omega_{y+1,j} - (y-1)\omega_{y,j}$ more spinning time.

According to Lemma 2, because $x > y+1$,

$$(x-1)\omega_{x,j} - (x-2)\omega_{x-1,j} \geq y\omega_{y+1,j} - (y-1)\omega_{y,j} \quad (20)$$

Next, if $x = 2$, after removing one request from a group of size 2, the removed request can only form a new size one group by itself. In this case, the estimated total spinning time cannot be increased.

Therefore, the estimated total spinning time cannot be increased, compared to the immediate prior state, after each individual necessary safe iteration. Consequently, after any number of necessary safe iterations, the new set of groups always causes no more total spinning time than the original state, which proves Theorem 1. ■

APPENDIX B.
MORE EXPERIMENTS

First, a $m = 4$ processor system is evaluated. The number of tasks in each taskset is set to 10, 15 or 25 in different tests. ψ^{bound}, CS^{lb} and CS^{ub} are always set to 5, 10 and 25 respectively.

Figures 59 to 61 depict the performance of all three concerned analyses under *DM*, *DCM* and *DkC* policies when each taskset consists of 10 tasks. As can be seen in these figures, *lp-CDW* generally performs better than *WIA* under all three priority assignment policies. However, when the total utilisation is low, *WIA* does outperform *lp-CDW* by a small difference under *DCM* and *DkC*. Nonetheless, *m-CDW* always performs better than the other two analyses regardless of the priority assignment policy chosen.

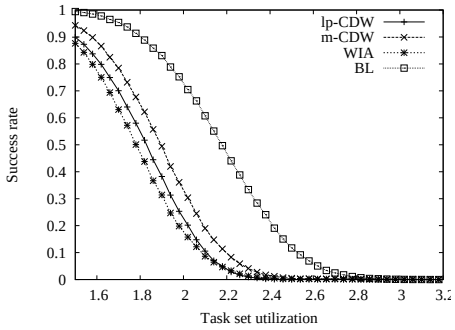

Figure 59. *m-CDW* vs *lp-CDW* vs *WIA* on 4 processors under DM when task number is 10

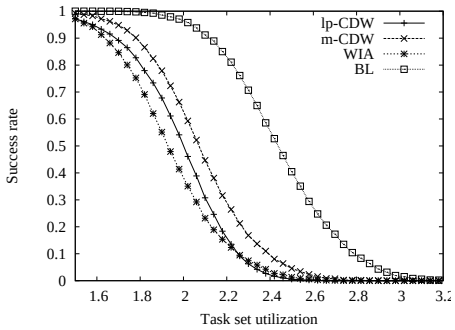

Figure 60. *m-CDW* vs *lp-CDW* vs *WIA* on 4 processors under DCM when task number is 10

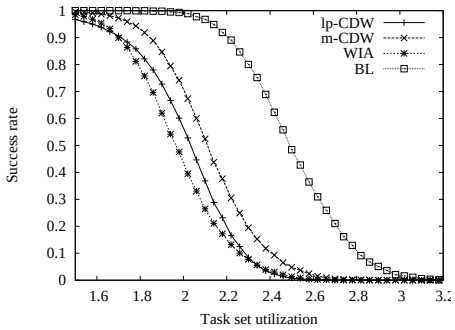

Figure 61. *m-CDW* vs *lp-CDW* vs *WIA* on 4 processors under DkC when task number is 10

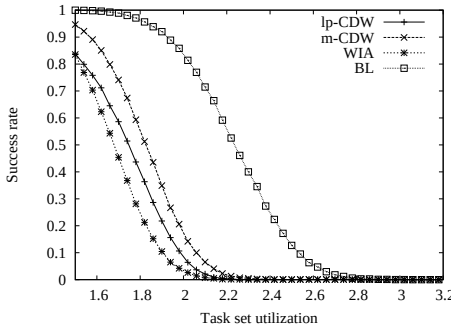

Figure 62. *m-CDW* vs *lp-CDW* vs *WIA* on 4 processors under DM when task number is 15

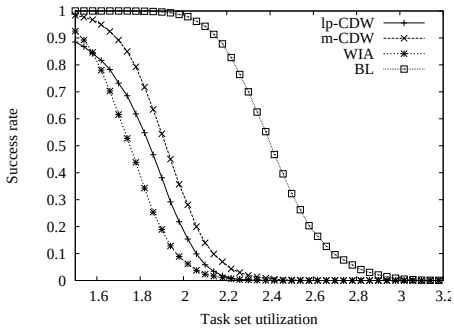

Figure 63. *m-CDW* vs *lp-CDW* vs *WIA* on 4 processors under DCM when task number is 15

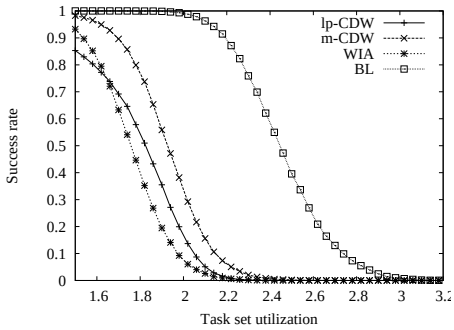

Figure 64. *m-CDW* vs *lp-CDW* vs *WIA* on 4 processors under DkC when task number is 15

When the task number in each taskset is raised to 15 (figures 62 to 64), the relative performance of *lp-CDW* compared against *WIA* degrades in the low utilisation cases, especially under the *DkC* priority assignment policy. However, the relative performance of *m-CDW* compared against the other two remains strong. The success rate gaps become even larger. This trend continues when the task number is increased to 25 (figures 65 to 67). Interestingly, the relative performance of *m-CDW* compared against *WIA* remains very good even when the *lp-CDW* analysis can barely recognise any schedulable tasksets by itself (figure 67).

As illustrated by figures 68 to 70, the priority assignment policy *DkC* generally has a better performance than the other two policies. However, the performance difference between *DkC* and *DCM* is insignificant if the task number in each taskset is no fewer than 15.

Next, we change the number of processors to $m = 8$ and keep all other parameters unmodified. As illustrated by figures 71 to 73, in the 10 task cases, the performance of all three analyses is very close under every priority assignment policy though *m-CDW* still dominates the other two. It is also clear the *DM* policy is not as good as the other two policies in terms of the absolute performance of all three analyses.

When the number of tasks is changed from 10 to 15 (figures 74 to 76), *lp-CDW* and *m-CDW* are more resilient to this change while the performance of *WIA* degrades more quickly. This trend continues when the task number is increased to 25 (figures 77 to 79).

As illustrated by figures 80 to 82, the priority assignment policy *DkC* always performs better than the other two policies for *m-CDW* in the 8 processor case. The performance of *DM* is particularly poor and *DCM* is only close to *DkC* when the number of tasks in each taskset is 10.

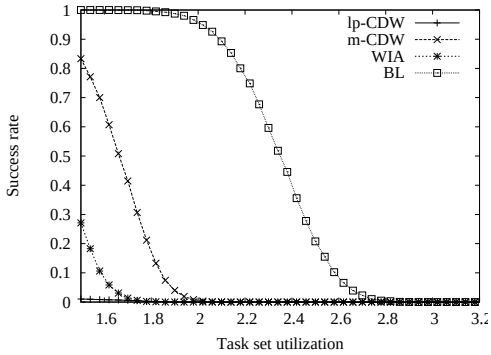

Figure 65. *m-CDW* vs *lp-CDW* vs *WIA* on 4 processors under DM when task number is 25

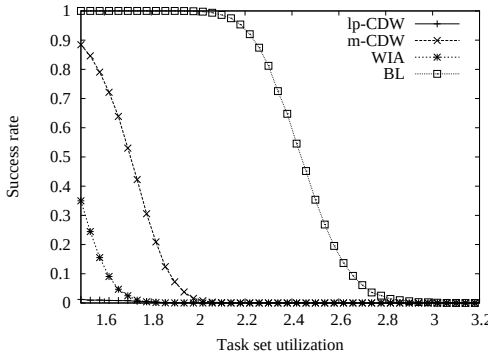

Figure 66. *m-CDW* vs *lp-CDW* vs *WIA* on 4 processors under DCM when task number is 25

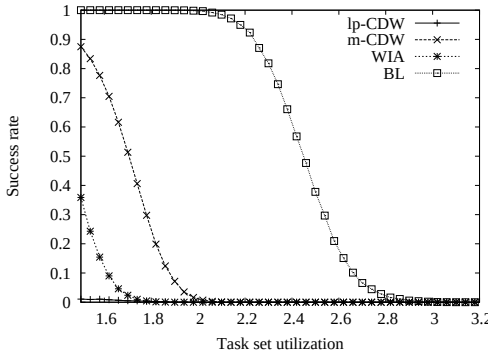

Figure 67. *m-CDW* vs *lp-CDW* vs *WIA* on 4 processors under DkC when task number is 25

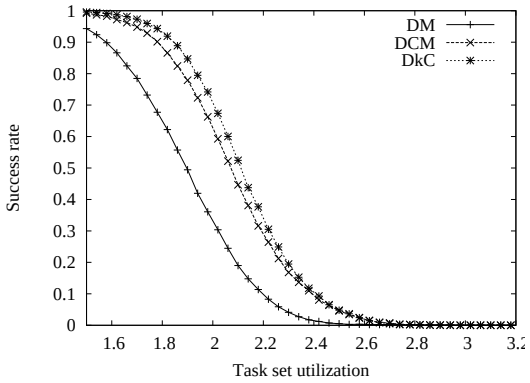

Figure 68. *m-CDW* on 4 processors under different priority assignment policies when task number is 10

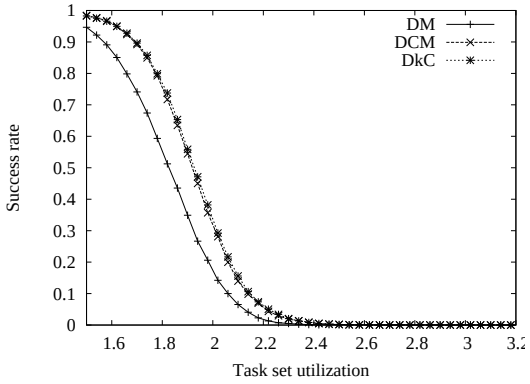

Figure 69. *m-CDW* on 4 processors under different priority assignment policies when task number is 15

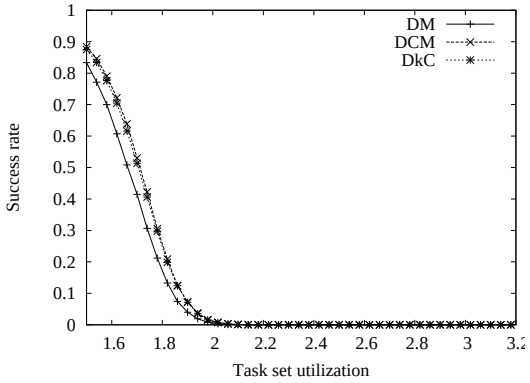

Figure 70. *m-CDW* on 4 processors under different priority assignment policies when task number is 25

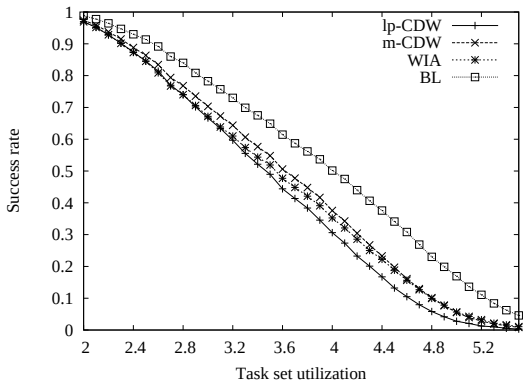

Figure 71. *m-CDW* vs *lp-CDW* vs *WIA* on 8 processors under DM when task number is 10

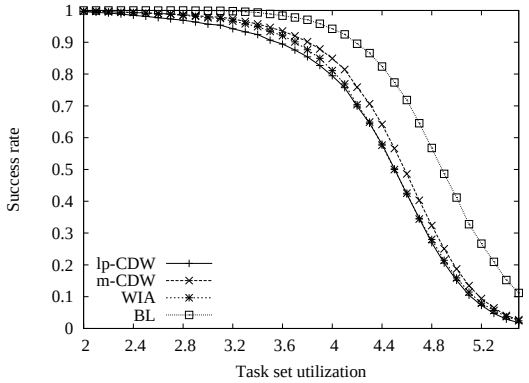

Figure 72. *m-CDW* vs *lp-CDW* vs *WIA* on 8 processors under DCM when task number is 10

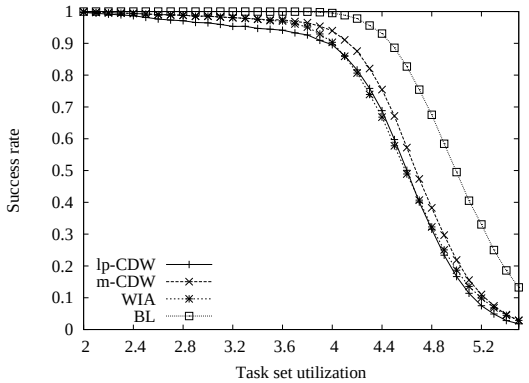

Figure 73. *m-CDW* vs *lp-CDW* vs *WIA* on 8 processors under DkC when task number is 10

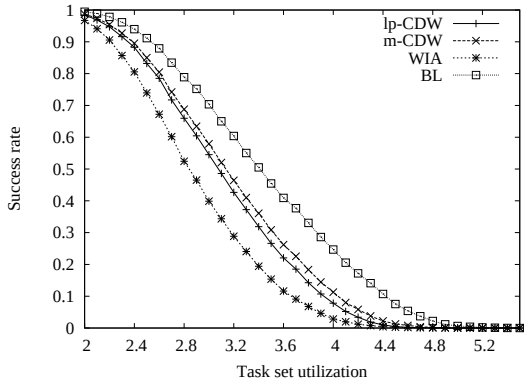

Figure 74. *m-CDW* vs *lp-CDW* vs *WIA* on 8 processors under DM when task number is 15

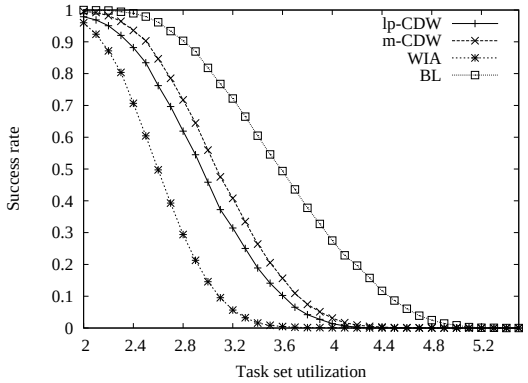

Figure 77. *m-CDW* vs *lp-CDW* vs *WIA* on 8 processors under DM when task number is 25

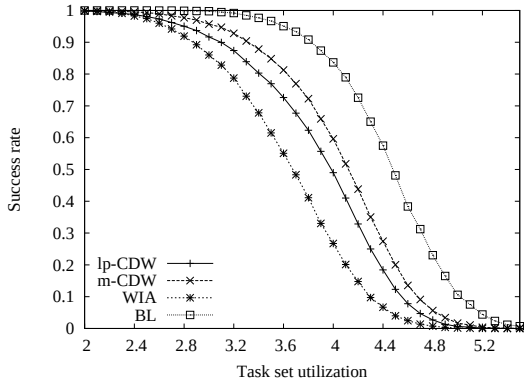

Figure 75. *m-CDW* vs *lp-CDW* vs *WIA* on 8 processors under DCM when task number is 15

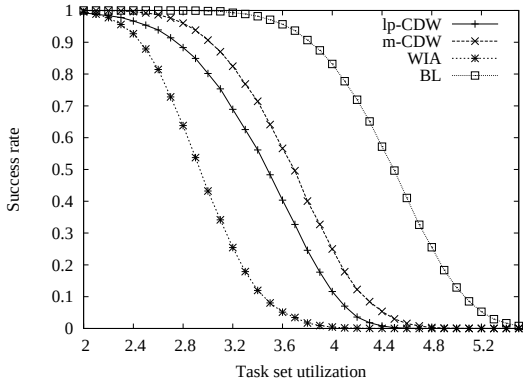

Figure 78. *m-CDW* vs *lp-CDW* vs *WIA* on 8 processors under DCM when task number is 25

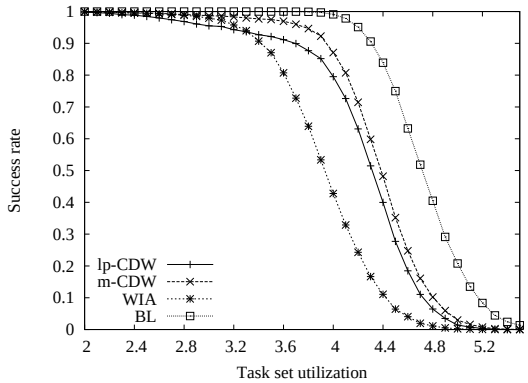

Figure 76. *m-CDW* vs *lp-CDW* vs *WIA* on 8 processors under DkC when task number is 15

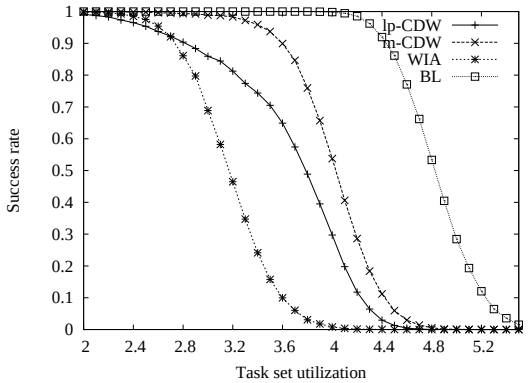

Figure 79. *m-CDW* vs *lp-CDW* vs *WIA* on 8 processors under DkC when task number is 25

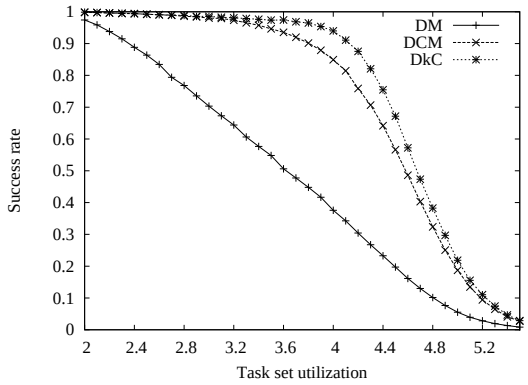

Figure 80. *m-CDW* on 8 processors under different priority assignment policies when task number is 10

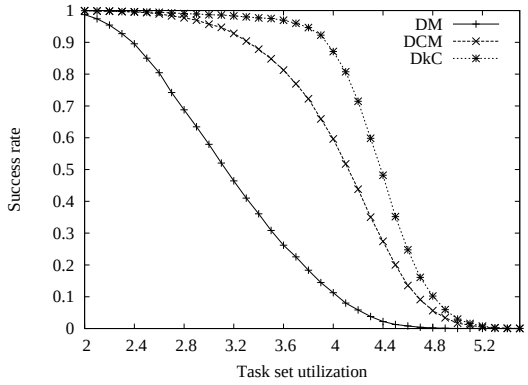

Figure 81. *m-CDW* on 8 processors under different priority assignment policies when task number is 15

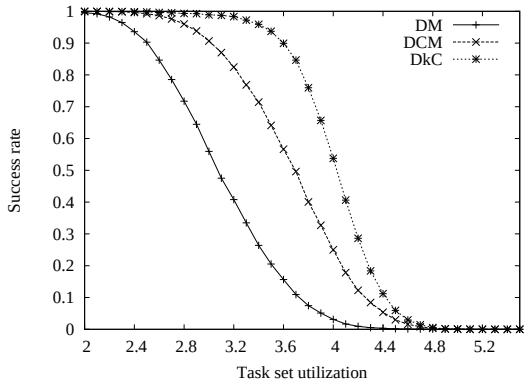

Figure 82. *m-CDW* on 8 processors under different priority assignment policies when task number is 25