Efficient Synchronization under Global EDF Scheduling on Multiprocessors *

UmaMaheswari C. Devi, Hennadiy Leontyev, and James H. Anderson
Department of Computer Science, The University of North Carolina at Chapel Hill

Abstract

We consider coordinating accesses to shared data structures in multiprocessor real-time systems scheduled under preemptive global EDF. To our knowledge, prior work on global EDF has focused only on systems of independent tasks. We take an initial step here towards a generic resource-sharing framework by considering simple shared objects, such as queues, stacks, and linked lists. In many applications, the predominate use of synchronization constructs is for sharing such simple objects. We analyze two synchronization methods for such objects, one based on queue-based spin locks and a second based on lock-free algorithms.

1 Introduction

In work on real-time systems, multiprocessor platforms (SMPs) are of growing importance. This is due to both hardware trends such as the emergence of multicore technologies and the prevalence of computationally-intensive applications for which single-processor designs are not sufficient.

In work on real-time multiprocessor systems, three basic scheduling approaches have been considered: *partitioning*, *Pfair-based global scheduling*, and *non-Pfair-based global scheduling*. Under partitioning, tasks are statically assigned to processors, and a uniprocessor scheduling algorithm is used on each processor to schedule its assigned tasks. In contrast, under global scheduling, a task may execute on any processor and may migrate across processors. Pfair scheduling [3] is currently the only known way of *optimally* scheduling recurrent real-time task systems on a multiprocessor. However, Pfair algorithms schedule tasks one quantum at a time, and as a result, jobs may be preempted and migrate across processors frequently. Such overheads can lower the amount of useful work accomplished.

Non-Pfair scheduling algorithms require restrictions on total utilization that can approach roughly 50% of the available processing capacity if every deadline is to be met [5]. However, recent work has shown that if bounded deadline tardiness is acceptable, then such restrictions can be lifted for global EDF [6, 18]. These results suggest that global EDF (henceforth referred to as just EDF) holds great promise as an effective scheduling method for real-time multiprocessor applications: if bounded tardiness is tolerable, then EDF enables applications to be deployed without the utilization restrictions and rigidity of partitioning schemes and without the potentially high implementation overheads of Pfair schemes. However, all prior work on EDF has pertained only to systems of independent tasks, *i.e.*, tasks that do not have synchronization constraints. Clearly, to be really useful, the current theory of EDF scheduling must be extended to properly address synchronization requirements. In this paper, we make an initial attempt to deal with such requirements by considering techniques for coordinating accesses to shared data structures.

Our particular focus in this paper is the *common case*, which we take to be non-nested accesses to simple shared objects. Research studies have been conducted that support the belief that simple objects are the predominate form of data sharing in many applications. For example, Tsigas and Zhang [16, 17] analyzed shared-memory applications from the SPLASH-2 benchmark suite [19] and Spark98 kernels suite [13] and found that almost all synchronization in these applications is for simple data structures such as buffers, queues, or stacks. Our conclusion that this is the common case is also supported anecdotally by discussions that the third author has had on this issue with many researchers over the years. Hence, we believe that it is appropriate to first consider supporting short, non-nested object accesses (or object calls) before enabling support for more complex objects, which will require more expensive approaches and extensive kernel support, such as priority inheritances and ceilings [14]. Two simple object-sharing approaches are therefore considered in this paper: queue-based spin locks and lock-free shared-object algorithms. Each is considered in turn below.

Queue locks. In the first approach we consider, shared object calls are implemented as critical sections accessed via queue-based spin locks (or, *queue locks*, for short) invoked within non-preemptive regions. In queue-lock algorithms, a task waits by busy-waiting, or spinning, on a "spin variable"(*i.e.*, continuously testing its status), and waiting tasks are ordered within a "spin queue" [12]. When a task attempts to acquire a lock, it appends its lock request onto the end of the spin queue and spins on a spin variable, which is exclusive to it. The task at the head of the queue may access the critical section, after which, it updates the spin variable for the next task in the queue so that it stops spinning. Queue locks have two interesting properties. First, busy-waiting is accomplished via *local spinning*: in other words, each task's spin variable is chosen such that it can be locally cached. This strategy ensures that waiting tasks do not generate excessive bus traffic. Second, lock requests are granted in FIFO order, and hence, bounds on waiting times can be easily determined. In particular, because we allow such locks to be invoked only within non-preemptive regions, if an object with an access cost

*Work supported by NSF grants CNS 0309825, CNS 0408996, and CCF 0541056. The first author was also supported by an IBM Ph.D. fellowship.

of at most e time units is shared by c tasks executing on an m-processor system, then a waiting task (that is, a task that has already initiated its request) can block for at most $(\min(m, c)-1)\cdot e$ time units before its critical section commences execution. In the common case under consideration, such waiting times will be short (as the e term here will be small). Note that if a task could be preempted within a critical section, then the waiting times for any tasks that it blocks could lengthen enormously. This is the case even with critical sections that are short in duration.

Though a FIFO lock-granting order may delay higher-priority jobs, the alternative, provided by a priority-based queue lock, has a higher time complexity and requires a more complicated implementation. Moreover, under EDF, introducing priorities within queue locks does not alter worst-case analysis. Similarly, disabling preemptions for tasks that are spinning may seem wasteful. However, the rationale behind spinning is that for short critical sections, waiting times would typically be much less than the overhead involved in blocking (as with semaphores) and later resuming a process. The solution proposed is in line with views expressed by others. For instance, in [20], the founder of RTLinux recommends accessing simple critical sections non-preemptively.

To enable the functionality discussed above under EDF, prior work on hard and soft real-time analysis under EDF must be extended to allow tasks to be comprised of preemptive and non-preemptive regions. An important contribution of this paper is to derive a tardiness bound under such a mixed preemptive/non-preemptive task model. The analysis used in doing this is based on that used previously by Devi and Anderson [6] in their work on EDF and non-preemptive EDF. In fact, the prior bounds presented by them are special cases of the bound presented here. Given these results, the impact of queue-lock overheads on tardiness can be easily assessed. Though our main focus in this paper is soft real-time systems, we also show how to account for queue-lock overheads under EDF in hard real-time systems. Finally, it is worth noting that unlike in the case of uniprocessors, a fully non-preemptive solution would not be effective in eliminating contention to shared resources on multiprocessors. Thus, a simpler solution than queue locks, with comparable losses due to synchronization, seems unlikely.

Lock-free synchronization. Because disabling preemptions requires kernel support, the approach discussed above may not be suitable in all contexts. As an alternative, we suggest using lock-free shared-object algorithms, which work particularly well for the kinds of simple objects considered in this paper. In such algorithms, operations on shared data structures are implemented using "retry loops": operations are optimistically attempted and retried until successful. (See [1] for an in-depth discussion of lock-free synchronization.) Retries are needed in the event that concurrent operations by different tasks *interfere* with each other. No kernel support is required. For lock-free objects to be usable in real-time systems, it is important that bounds on interferences (and hence, retries) be determined.

While the viability of the lock-free approach for *uniprocessor*-based real-time systems is well-known [1], on *multiprocessors*, lock-free sharing is often considered impractical, because of difficulties in computing reasonable retry bounds. Nevertheless, this approach may be a reasonable option in the absence of kernel support. Hence, as a second contribution, we show how to bound the number of interferences in hard and soft real-time systems scheduled under EDF.

Performance evaluation. As a third contribution, we present an evaluation of the two approaches discussed above using randomly-generated task systems. In this evaluation, we determined the loss in total system utilization and the increase to tardiness bounds for tasks. Execution times for operations on shared objects when generating random tasks were chosen based on experimental data collected from a real test bed. We found that, for simple objects, access times are very short, usually in the 1-5μs range. By extrapolating from these results, we concluded that retry costs in lock-free implementations would also be small. When considering randomly-generated tasks, such short access times were seen to have only a low impact when the number of shared-object invocations per task is at most three. We believe this to be a reasonable upper bound on the degree of contention in many applications. This is supported by Lamport [10], who noted that contention for critical sections is rare in a well-designed system. Hence, our overall conclusion is that, in the common case, object-sharing support can be provided under EDF without complicated techniques and with little overhead.

Related work. As mentioned earlier, to our knowledge, task synchronization under global EDF has not been considered in any prior work. The problem has been addressed in some other multiprocessor scheduling approaches, though. In [14], an extension of the *priority-ceiling protocol*, which is for use with semaphores, has been presented for partitioned rate-monotonic scheduling. Semaphore-based approaches and the lock-free approach have been considered for Pfair-scheduled systems in [8] and [9], respectively. For use in multiprocessor-based real-time kernels, an approach based on spin locks has been proposed in [15]. However, this work is mainly concerned with lowering interrupt latencies, and not with the schedulability of a real-time workload, and the evaluation is entirely empirical.

The rest of this paper is organized as follows. Our system model is described in Sec. 2. A tardiness bound for tasks with non-preemptive code segments is derived in Sec. 3. An analysis of EDF with the two proposed synchronization approaches for both hard and soft real-time systems is presented in Sec. 4. Sec. 5 presents the above-mentioned experimental evaluation, and finally, Sec. 6 concludes.

2 System Model and Notation

Towards determining a tardiness bound for tasks synchronized using queue locks, we consider the scheduling of a *sporadic task system* τ comprised of $n \geq 2$ *sporadic tasks* denoted $T_1, \ldots, T_n$ on $m \geq 2$ identical processors. Each sporadic task T_i is characterized by a tuple (e_i, p_i), where p_i is the *minimum inter-arrival time* between two consecutive jobs of T_i (also known as its *period*) and e_i is the *worst-case execution cost* of each job of T_i. The kth job of T_i is denoted $T_{i,k}$. The release time of $T_{i,k}$ is denoted $r_{i,k}$. p_i is also the *relative deadline* of each job of T_i.

The *absolute deadline* (or just, deadline) of $T_{i,k}$ is denoted $d_{i,k}$ $(= r_{i,k} + p_i)$.

Each job of each task T_i may be comprised of zero or more non-preemptable code segments. The maximum execution cost of any such segment of any task is denoted $b_{\max}$.[1] (As noted earlier, it is our intention to invoke queue-lock algorithms within such segments. In this case, $b_{\max}$ will depend on spinning times. We consider this issue later in Sec. 4.) The *utilization* of T_i is given by $u_i = e_i/p_i$. The *total utilization* of τ is defined as $U_{sum}(\tau) = \sum_{i=1}^{n} u_i$. The minimum execution cost of any task is denoted $e_{\min}(\tau)$.

A sporadic task system τ is *concrete* if the release time of every job of each of its tasks is specified, and *non-concrete*, otherwise. The type of the task system is specified only when necessary. The results in this paper are for non-concrete task systems, and hence hold for every concrete task system.

In *soft* real-time systems, jobs may miss their deadlines. The *tardiness of a job* $T_{i,j}$ in a schedule $\mathcal{S}$ is defined as $tardiness(T_{i,j}, \mathcal{S}) = \max(0, t - d_{i,j})$, where t is the time at which $T_{i,j}$ completes executing in $\mathcal{S}$. If κ is the maximum tardiness of any job of any task of any feasible task system under scheduling algorithm $\mathcal{A}$, then $\mathcal{A}$ is said to *ensure a tardiness bound of* κ. We assume that *deadline misses do not delay future job releases*. That is, even if a job misses its deadline, the release time of the next job of that task remains unaltered.

We will refer to the EDF algorithm that is cognizant of the non-preemptive sections of a task and executes them non-preemptively as EDF-hybrid. (In a real implementation, special system calls will be required to inform the scheduler when a non-preemptive section is entered and exited.) At any time, higher priority is accorded to jobs with earlier deadlines, subject to not preempting a job that is executing in a non-preemptive section. Ties are resolved arbitrarily. A job executing in a preemptive section may be preempted by an arriving higher-priority job and may later resume execution on a different processor. Note that fully-preemptive and non-preemptive EDF are special cases of EDF-hybrid.

The tardiness bound we derive is expressed in terms of the highest task execution costs and utilizations, and the total system utilization. To express the bound easily, we let ϵ_i (resp., μ_i) denote the ith execution cost (resp., task utilization) in non-increasing order.[2] Λ is defined as follows.

$$\Lambda = \begin{cases} U_{sum}(\tau) - 1, & U_{sum}(\tau) \text{ is integral} \\ \lfloor U_{sum}(\tau) \rfloor, & \text{otherwise} \end{cases} \tag{1}$$

3 A Tardiness Bound for EDF-hybrid

In this section, we derive a tardiness bound for EDF-hybrid. Our approach is the same as that used in [6] in deriving tardiness bounds under fully-preemptive and non-preemptive EDF. Our contribution here is in integrating the two derivations and in deriving a bound that is dependent on not just the individual task parameters but also on total system utilization.

[1] In fact, tasks with the shortest relative deadline can be excluded in determining $b_{\max}$, since such tasks cannot block any task under EDF.

[2] ϵ_i and μ_i may not correspond to T_i and may not represent the parameters of the same task.

The derivation involves comparing the allocations to a concrete task system τ in a processor sharing (PS) schedule and an EDF-hybrid schedule, and quantifying the difference between the two. In a PS schedule, each job of T_i is allocated a fraction u_i of a processor at each instant in the interval between its release time and deadline. Because $U_{sum} \leq m$ holds and relative deadlines equal periods, the total demand at any instant will not exceed m in a PS schedule, and every job will complete executing exactly at its deadline.

3.1 Definitions

The system start time is assumed to be zero. For any time $t > 0$, t^- denotes the time $t - \epsilon$ in the limit $\epsilon \to 0+$.

Definition 1 (active tasks and active jobs): A task T_i is said to be *active* at time t if there exists a job $T_{i,j}$ (called T_i's *active job* at t) such that $r_{i,j} \leq t < d_{i,j}$. By our task model, every task can have at most one active job at any time.

Definition 2 (pending jobs): $T_{i,j}$ is said to be *pending* at t in a schedule $\mathcal{S}$ if $r_{i,j} \leq t$ and $T_{i,j}$ has not completed execution by t in $\mathcal{S}$. Note that a job with a deadline at or before t is not considered to be active at t even if it is pending at t.

Definition 3 (ready jobs): A pending job $T_{i,j}$ is said to be *ready* at t in a schedule $\mathcal{S}$ if $t \geq r_{i,j}$ and all prior jobs of T_i have completed execution by t in $\mathcal{S}$.

Let $\mathsf{A}(\mathcal{S}, T_i, t_1, t_2)$ denote the total time allocated to T_i in an arbitrary schedule $\mathcal{S}$ for τ in $[t_1, t_2)$. Then, since in PS_τ (the PS schedule for τ), T_i is allocated a fraction u_i at each instant it is active in $[t_1, t_2)$, we have $\mathsf{A}(\mathsf{PS}_\tau, T_i, t_1, t_2) \leq (t_2 - t_1)u_i$. The total allocation to τ in the same interval in PS_τ is

$$\mathsf{A}(\mathsf{PS}_\tau, \tau, t_1, t_2) \leq \sum_{T_i \in \tau} (t_2 - t_1)u_i = U_{sum}(\tau) \cdot (t_2 - t_1). \tag{2}$$

The difference between the total allocations to T_i up to time t in PS_τ and an arbitrary schedule $\mathcal{S}$ is defined as the *lag of task* T_i *at time* t *in schedule* $\mathcal{S}$, and is given by

$$\mathsf{lag}(T_i, t, \mathcal{S}) = \mathsf{A}(\mathsf{PS}_\tau, T_i, 0, t) - \mathsf{A}(\mathcal{S}, T_i, 0, t). \tag{3}$$

The *total lag of a task system* τ *at* t, denoted $\mathsf{LAG}(\tau, t, \mathcal{S})$, is given by

$$\begin{aligned} \mathsf{LAG}(\tau, t, \mathcal{S}) &= \sum_{T_i \in \tau} \mathsf{lag}(T_i, t, \mathcal{S}) \\ &= \mathsf{A}(\mathsf{PS}_\tau, \tau, 0, t) - \mathsf{A}(\mathcal{S}, \tau, 0, t). \end{aligned} \tag{4}$$

Note that $\mathsf{LAG}(\tau, 0, \mathcal{S})$ and $\mathsf{lag}(T_i, 0, \mathcal{S})$ are both zero, and that by (3) and (4), we have the following for $t_2 > t_1$.

$$\begin{aligned} \mathsf{lag}(T_i, t_2, \mathcal{S}) &= \mathsf{lag}(T_i, t_1, \mathcal{S}) + \\ &\quad \mathsf{A}(\mathsf{PS}_\tau, T_i, t_1, t_2) - \mathsf{A}(\mathcal{S}, T_i, t_1, t_2) \\ \mathsf{LAG}(\tau, t_2, \mathcal{S}) &= \mathsf{LAG}(\tau, t_1, \mathcal{S}) + \\ &\quad \mathsf{A}(\mathsf{PS}_\tau, \tau, t_1, t_2) - \mathsf{A}(\mathcal{S}, \tau, t_1, t_2) \end{aligned} \tag{5}$$

Lag for jobs. The notion of lag defined above for tasks and task sets can be applied to jobs and job sets in an obvious manner. Let τ denote a concrete task system, and Ψ a subset of jobs in τ. Let $\mathsf{A}(\mathsf{PS}_\tau, T_{i,j}, t_1, t_2)$ and $\mathsf{A}(\mathcal{S}, T_{i,j}, t_1, t_2)$ denote the allocations to $T_{i,j}$ in $[t_1, t_2)$ in PS_τ and $\mathcal{S}$, respectively. Then, $\mathsf{lag}(T_{i,j}, t, \mathcal{S}) = \mathsf{A}(\mathsf{PS}_\tau, T_{i,j}, r_{i,j}, t) - \mathsf{A}(\mathcal{S}, T_{i,j}, r_{i,j}, t)$,

and $\mathsf{LAG}(\Psi, t, \mathcal{S}) = \sum_{T_{i,j} \in \Psi} \mathsf{lag}(T_{i,j}, t, \mathcal{S})$. The total allocation in $[0, t)$, where $t > 0$, to a job that is neither pending at t^- in $\mathcal{S}$ nor is active at t^- is the same in both $\mathcal{S}$ and PS_τ, and hence, its lag at t is zero. Therefore, for $t > 0$, we have

$$\mathsf{LAG}(\Psi, t, \mathcal{S}) = \sum_{\substack{\{T_{i,j} \text{ is in } \Psi, \text{ and is pending} \\ \text{or active at } t^-\}}} \mathsf{lag}(T_{i,j}, t, \mathcal{S}).$$

The above can be rewritten using task lags as follows (since no job can be scheduled before its release time).

$$\mathsf{LAG}(\Psi, t, \mathcal{S}) \leq \sum_{\substack{\{T_i \in \tau \,:\, T_{i,j} \text{ is in } \Psi, \text{ and is} \\ \text{pending or active at } t^-\}}} \mathsf{lag}(T_i, t, \mathcal{S}) \tag{6}$$

The total utilization of Ψ at time t is defined as the sum of the utilizations of tasks with an active job at t in Ψ:

$$U_{sum}(\Psi, t) = \sum_{\substack{\{T_i \in \tau \,:\, T_{i,j} \text{ is in } \Psi \text{ and is ac-} \\ \text{tive at } t\}}} u_i \leq U_{sum}(\tau). \tag{7}$$

The counterparts of (2) and (5) for job sets are as follows.

$$\mathsf{A}(\mathsf{PS}_\tau, \Psi, t_1, t_2) = \int_{t_1}^{t_2} U_{sum}(\Psi, t)\mathrm{dt} \leq (t_2 - t_1) \cdot U_{sum}(\tau) \tag{8}$$

$$\mathsf{LAG}(\Psi, t_2, \mathcal{S}) = \mathsf{LAG}(\Psi, t_1, \mathcal{S}) + \mathsf{A}(\mathsf{PS}_\tau, \Psi, t_1, t_2) - \mathsf{A}(\mathcal{S}, \Psi, t_1, t_2) \tag{9}$$

Definition 4 (busy and non-busy intervals): A time interval $[t_1, t_2)$, where $t_2 > t_1$, is said to be *busy* for Ψ if all m processors are executing some job in Ψ at each instant in the interval, *i.e.*, no processor is ever idle in the interval or executes a job not in Ψ. An interval $[t_1, t_2)$ that is not busy for Ψ is said to be *non-busy* for Ψ, and is *maximally non-busy* if every time instant in $[t_1, t_2)$ is non-busy, and either $t_1 = 0$ or t_1^- is busy.

If at least $U_{sum}(\Psi, t)$ tasks are executing their jobs in Ψ at any instant t in $[t_1, t_2)$ in a schedule $\mathcal{S}$, then the total allocation in $\mathcal{S}$ to jobs in Ψ is at least the allocation that Ψ receives in a PS schedule. Therefore, by (9), the LAG of Ψ at t_2 cannot exceed that at t_1, and we have the following lemma.

Lemma 1 *If* $\mathsf{LAG}(\Psi, t + \delta, \mathcal{S}) > \mathsf{LAG}(\Psi, t, \mathcal{S})$*, where* $\delta > 0$ *and* $\mathcal{S}$ *is a schedule for* τ*, then* $[t, t + \delta)$ *is a non-busy interval for* Ψ*. Furthermore, there exists at least one instant* t' *in* $[t, t + \delta)$ *at which fewer than* $U_{sum}(\Psi, t')$ *tasks are executing their jobs in* Ψ*.*

3.2 Deriving a Tardiness Bound

Given an arbitrary non-concrete task system τ^N, we are interested in determining the maximum tardiness of any job of any task in any concrete instantiation of τ^N. Let τ be a concrete instantiation of τ^N, $T_{\ell,j}$ a job in τ, $t_d = d_{\ell,j}$, and $\mathcal{S}$ an $\mathsf{EDF\text{-}hybrid}$ schedule for τ with the following property.

(P) The tardiness of every job of every task T_k in τ with deadline less than t_d is at most $x + e_k$ in $\mathcal{S}$, where $x \geq 0$.

Then, determining the smallest x, independent of the parameters of T_ℓ, such that the tardiness of $T_{\ell,j}$ remains at most $x + e_\ell$ would by induction imply a tardiness of at most $x + e_k$ for all jobs of every task T_k in τ. Because τ is arbitrary, the tardiness bound will hold for every concrete instance of τ^N.

Our objective is easily met if $T_{\ell,j}$ completes by its deadline, t_d, so assume otherwise. The completion time of $T_{\ell,j}$ depends on the amount of work that can compete with $T_{\ell,j}$ after t_d. Hence, we follow the steps below to determine x.

(S1) Compute an upper bound (UB) on the work (including that due to $T_{\ell,j}$) that can compete with $T_{\ell,j}$ after t_d.

(S2) Determine a lower bound (LB) on the amount of such work required for the tardiness of $T_{\ell,j}$ to exceed $x + e_\ell$.

(S3) Determine the smallest x such that the tardiness of $T_{\ell,j}$ is at most $x + e_\ell$ using UB and LB.

Let Ψ and $\overline{\Psi}$ be defined as follows.

$$\Psi \stackrel{\text{def}}{=} \text{set of all jobs with deadlines at most } t_d \text{ of tasks in } \tau$$

$$\overline{\Psi} \stackrel{\text{def}}{=} \text{set of all jobs of } \tau \text{ that are not in } \Psi$$

(*i.e.*, jobs with deadlines later than t_d)

Under $\mathsf{EDF\text{-}hybrid}$, competing work for $T_{\ell,j}$ at t_d is given by **(i)** the amount of work pending at t_d for jobs in Ψ plus **(ii)** the amount of work demanded after t_d by non-preemptive sections of jobs that are not in Ψ but that commenced execution before t_d. Because the deadline of every job in Ψ is at most t_d, the first component is given by $\mathsf{LAG}(\Psi, t_d, \mathcal{S})$. To facilitate computing the second component, which will be denoted $\mathsf{B}(\tau, \Psi, t_d, \mathcal{S})$, we define the following.

Definition 5 (priority inversions, blocking jobs, and blocked jobs): Under $\mathsf{EDF\text{-}hybrid}$, a *priority inversion* occurs when a ready, higher-priority job waits while one or more lower-priority jobs execute in non-preemptive sections. Under such scenarios, the waiting higher-priority jobs are said to be *blocked* jobs, while executing lower-priority jobs are said to be *blocking* jobs. Note that a pending, higher-priority job is not considered blocked unless it is ready (*i.e.*, no prior job of the same task is pending).

Recall that in a non-busy interval for Ψ, fewer than m jobs from Ψ execute. In an $\mathsf{EDF\text{-}hybrid}$ schedule, such a non-busy interval for Ψ can be classified into two types depending on whether a job in $\overline{\Psi}$ is executing while a ready job from Ψ is waiting. We will refer to the two types as *blocking* and *non-blocking* non-busy intervals. A *blocking, non-busy interval* is one in which a job in $\overline{\Psi}$ is executing while a ready job from Ψ is waiting, whereas a *non-blocking, non-busy interval* is one in which fewer than m jobs from Ψ are executing, but there does not exist a ready job in Ψ that is waiting. Definitions of maximal versions of these intervals are analogous to that of a maximally non-busy interval given in Def. 4.

Definition 6 (pending blocking jobs ($\mathcal{B}$) and work (B)): The set of all jobs in $\overline{\Psi}$ that commence executing a non-preemptive section before t and may continue to execute the same non-preemptive section at t in $\mathcal{S}$ is denoted $\mathcal{B}(\tau, \Psi, t, \mathcal{S})$ and the total

amount of work pending at t for such non-preemptive sections is denoted $\mathsf{B}(\tau, \Psi, t, \mathcal{S})$.

We now determine an upper bound on the sum of the two components of the competing work described above, *i.e.*, $\mathsf{LAG}(\Psi, t_d, \mathcal{S}) + \mathsf{B}(\tau, \Psi, t_d, \mathcal{S})$ (step (S1)).

3.2.1 Upper Bound on $\mathsf{LAG}(\Psi, t_d, \mathcal{S}) + \mathsf{B}(\tau, \Psi, t_d, \mathcal{S})$

By Lemma 1, the LAG of Ψ can increase only across a non-busy interval for Ψ. Similarly, note that if $\mathsf{B}(\tau, \Psi, t_d, \mathcal{S})$ is non-zero, then one or more jobs in $\overline{\Psi}$ should be executing in non-preemptive sections at t_d^-, that is, t_d^- should be a non-busy instant. Hence, to determine an upper bound on the value that we are seeking, it suffices to consider only non-busy intervals in $[0, t_d)$. By reasoning about the number of tasks that can execute in and just before a non-busy interval and their lags, the following lemma (proved in an appendix) can be shown to hold.

Lemma 2 $\mathsf{LAG}(\Psi, t_d, \mathcal{S}) + \mathsf{B}(\tau, \Psi, t_d, \mathcal{S}) \leq (\sum_{i=1}^{\Lambda}(x \cdot \mu_i + \max(\epsilon_i, b_{\max}))) + (m - \Lambda) \cdot b_{\max}$.

3.2.2 Lower Bound on $\mathsf{LAG} + \mathsf{B}$ (Step (S2))

Lemma 3 *If* $\mathsf{LAG}(\Psi, t_d, \mathcal{S}) + \mathsf{B}(\tau, \Psi, t_d, \mathcal{S}) \leq mx + e_\ell$*, then,* $tardiness(T_{\ell,j}, \mathcal{S})$ *is at most* $x + e_\ell$*.*

Proof: To prove the lemma, we show that $T_{\ell,j}$ completes executing by $t_d + x + e_\ell$. If $j > 1$, then $d_{\ell,j-1} \leq t_d - p_\ell$ holds, and hence by (P), we have the following.

(R) $T_{\ell,j-1}$ completes executing by $t_d + x + e_\ell - p_\ell$, for $j > 1$.

Let $\delta_\ell < e_\ell$ denote the amount of time that $T_{\ell,j}$ has executed for before time t_d. Then, the amount of work pending for $T_{\ell,j}$ at t_d is $e_\ell - \delta_\ell$. Recall that the total amount of work pending at t_d for jobs in Ψ and the non-preemptive sections of jobs in $\overline{\Psi}$ that commenced execution before t_d is given by $\mathsf{LAG}(\Psi, t_d, \mathcal{S}) + \mathsf{B}(\tau, \Psi, t_d, \mathcal{S})$, which, by the statement of the lemma, is at most $mx + e_\ell$. Let $y = x + \delta_\ell/m$. At the risk of abusing terms, let a time interval after t_d in which each processor is busy executing a job of Ψ or that non-preemptive part of a job in $\mathcal{B}(\tau, \Psi, t_d, \mathcal{S})$ that commenced execution before t_d be referred to as a busy interval. We consider the following two cases.

Case 1: $[t_d, t_d + y)$ is busy. In this case, the amount of work completed in $[t_d, t_d + y)$ is exactly $my = mx + \delta_\ell$, and hence, the amount of work pending at $t_d + y$ for jobs in Ψ and the non-preemptive sections of jobs in $\overline{\Psi}$ that commenced execution before t_d is at most $mx + e_\ell - (mx + \delta_\ell) = e_\ell - \delta_\ell$. Hence, if $T_{\ell,j}$ does not execute in $[t_d, t_d + y)$, then this pending work corresponds to that of $T_{\ell,j}$. Note that $T_{\ell,j}$ cannot be preempted once it commences execution after t_d. Thus, the latest time that $T_{\ell,j}$ resumes (or begins, if $\delta_\ell = 0$) execution after t_d is $t_d + y$, and hence, $T_{\ell,j}$ completes execution at or before $t_d + y + e_\ell - \delta_\ell \leq t_d + x + e_\ell$.

Case 2: $[t_d, t_d + y)$ is not busy. Let t' denote the first non-busy instant in $[t_d, t_d + y)$. By the definition of a busy interval used in this lemma, this implies the following.

(J) In $[t_d, t')$, no job in $\overline{\Psi}$ executes in a non-preemptive section that did not commence before t_d or in a preemptive section, and at most $m - 1$ tasks have jobs in Ψ or non-preemptive sections that commenced before t_d pending at or after t'.

Hence, no job of Ψ can be blocked by a job in $\overline{\Psi}$ at or after t'. Therefore, if $T_{\ell,j}$ has not completed executing before $t_d + y$, then either $T_{\ell,j}$ or a prior job of T_ℓ should be executing at t'. If $T_{\ell,j}$ is executing at t', then because $t' < t_d + y$ holds, $T_{\ell,j}$ will complete executing before $t_d + y + e_\ell - \delta_\ell \leq t_d + x + e_\ell$. The remaining possibility is that $j > 1$ holds, and that a job of T_ℓ that is prior to $T_{\ell,j}$ is executing at t'. In this case, $T_{\ell,j}$ could not have executed before t_d, and hence $\delta_\ell = 0$ and $y = x$ holds. Thus, $t' < t_d + y = t_d + x$ holds. Let t_c denote the time at which $T_{\ell,j-1}$ completes executing. Then, by (R), $t_c \leq t_d - p_\ell + x + e_\ell \leq t_d + x$ holds. If $t_c \geq t'$ holds, then by (J), $T_{\ell,j}$ can commence execution at $t_c \leq t_d + x$ (on the same processor as that on which $T_{\ell,j-1}$ executed), and hence, can complete executing by $t_d + x + e_\ell$. On the other hand, if $t_c < t'$ holds, then because $T_{\ell,j-1}$ completes execution at t_c, $T_{\ell,j}$ is ready at t'. Therefore, by (J), $T_{\ell,j}$ cannot be blocked at t'. Hence, $T_{\ell,j}$ commences execution at $t' < t_d + y = t_d + x$ and completes executing by $t_d + x + e_\ell$. ■

3.2.3 Finishing Up (Step (S3))

Solving for x using the upper bound on $\mathsf{LAG} + \mathsf{B}$ given by Lemma 2 and the lower bound on the same quantity, as given by Lemma 3, required for tardiness of $T_{\ell,j}$ to exceed $x + e_\ell$, *i.e.*, solving for x in $(\sum_{i=1}^{\Lambda}(x \cdot \mu_i + \max(\epsilon_i, b_{\max}))) + (m - \Lambda) \cdot b_{\max} \leq mx + e_\ell$ yields

$$x \geq \frac{(\sum_{i=1}^{\Lambda} \max(\epsilon_i, b_{\max})) + (m - \Lambda) \cdot b_{\max} - e_\ell}{m - \sum_{i=1}^{\Lambda} \mu_i}. \quad (10)$$

Hence, if x equals the right-hand side of the above inequality, then the tardiness of $T_{\ell,j}$ would not exceed $x + e_\ell$. A value of x that is independent of the parameters of T_ℓ is obtained by replacing e_ℓ by $e_{\min}$ in (10). By inducting over the jobs of τ in deadline order, we have the following theorem.

Theorem 1 *Let* τ *be as defined earlier. Let* $U_{sum}(\tau) \leq m$ *and let* $b_{\max}$ *be the maximum length of any non-preemptive section of any task in* τ*. Then,* EDF-hybrid *ensures a tardiness of at most* $x + e_k$ *to every task* T_k *in* τ*, where*

$$x = \frac{(\sum_{i=1}^{\Lambda} \max(\epsilon_i, b_{\max})) + (m - \Lambda) \cdot b_{\max} - e_{\min}}{m - \sum_{i=1}^{\Lambda} \mu_i}.$$

4 Analysis with Synchronization

We now show how to extend the analysis of EDF to synchronization with the queue-lock and lock-free approaches.

4.1 Analysis with Queue Locks

Under queue-lock-based synchronization, a job may be subject to blocking under two scenarios: **(i)** the job is executing and requires access to a resource for which one or more jobs have already enqueued their requests onto the spin queue, or **(ii)** the job

becomes ready when one or more lower-priority jobs (with later deadlines) are in their non-preemptive sections, either spinning or executing a critical section, and no processor is available. The former lengthens the amount of time a job spins on a lock. Hence, to account for its effect, the execution cost of each critical section has to be increased by the maximum amount of time for which entry into the critical section can be delayed after request for the lock is initiated. As explained in the introduction, computing this is straightforward. Also, the worst-case execution cost of each T_i has to be increased by the cumulative increase in the execution costs of all its critical sections. We denote the inflated execution cost of T_i as $e_i^{(s)}$. The overhead of non-preemptivity during spinning and in critical-section execution is accounted for by using the inflated critical-section execution costs in schedulability tests and tardiness-bound computations, as explained below.

Analysis for soft real-time. A tardiness bound for τ with shared resources is obtained by using $e_i^{(s)}$ instead of e_i and $u_i^{(s)} = e_i^{(s)}/p_i$ instead of u_i, for all i, and the maximum inflated execution cost of any critical section of any task for $b_{\max}$ in Theorem 1. Note that bounded tardiness on m processors can be guaranteed only if $e_i^{(s)} \leq p_i$ holds for all i and $\sum_{i=1}^{n} u_i^{(s)} \leq m$.

Analysis for hard real-time. Under EDF, and hence under EDF-hybrid, a task with a longer relative deadline cannot be blocked by a task with a shorter or equal relative deadline [11]. Thus, if b_i denotes the maximum execution cost of any non-preemptable segment of T_i, and tasks are ordered in the non-decreasing order of their relative deadlines, then each job of T_i can be blocked for at most $B_i = \max_{i+1 \leq j \leq n} b_j$ time units. Furthermore, since jobs may not miss deadlines and relative deadlines equal periods, no job can execute after the next job of the same task is released. Hence, each job J can be blocked, by a lower-priority job executing in its non-preemptive section, as soon as J is released only (*i.e.*, only at the beginning of its period). Therefore, guaranteeing that, under preemptive EDF, each job of each task T_k can be allocated e_k units of time in the interval $[a + B_k, a + p_k)$, where a is the arrival time of the job and B_k is as defined above, is sufficient to ensure that all deadlines will be met when τ is scheduled under EDF-hybrid. Hence, a simple sufficient schedulability test for EDF-hybrid is given by the following. For each τ, comprised of tasks with non-preemptable code segments, let τ' be defined as follows. $\tau' \stackrel{\text{def}}{=} \{T'_1, \cdots, T'_n\}$, where $T'_i \stackrel{\text{def}}{=} (e_i, p_i - B_i)$. If $p_i - B_i \geq e_i$, for all T_i, and τ' is schedulable under EDF, then τ is schedulable under EDF-hybrid. Schedulability under EDF can be determined using tests provided in [7, 2, 4]. When analyzing with queue locks, as with soft real-time analysis above, $e_i^{(s)}$ should be used instead of e_i.

4.2 Analysis with Lock-Free Synchronization

With lock-free synchronization, a bound on the number of interferences that can lead to failed retry loop iterations in a given interval of time needs to be determined. However, note that, unlike in the case of queue locks, under the lock-free approach, tasks do not block and there are no non-preemptive segments. To aid in stating results, we first define the following.

$r_{\max} \stackrel{\text{def}}{=}$ maximum execution cost of a single iteration of any retry loop of any task

$\gamma_i \stackrel{\text{def}}{=}$ set of all tasks that share one or more objects with T_i

$\beta_{i,k} \stackrel{\text{def}}{=}$ number of retry loops of T_k that access the same object as accessed by some retry loop of T_i

$\delta_i \stackrel{\text{def}}{=}$ a tardiness bound for T_i in the absence of interferences to any retry loop of any task

$\delta_i^I \stackrel{\text{def}}{=}$ a tardiness bound for T_i in the presence of interferences

A bound on the number of interferences to any job of T_i in a given interval of time is given by the following.

Claim 1 *For any task T_i, the total number of times that all retry loops combined of any of its jobs can fail in an interval $[t, t + \Delta)$ is at most $\sum_{T_k \in \gamma_i} (\lceil \frac{\Delta + \delta_k^I}{p_k} \rceil + 1) \cdot \beta_{i,k}$.*

Proof: The number of interferences to T_i due to T_k in an interval of length Δ depends on the number of jobs of T_k that can potentially execute in such an interval. The deadline of the earliest job of T_k that can execute in $[t, t + \Delta)$ is after $t - \delta_k^I$ (because tardiness for T_k is at most δ_k^I), and hence, its release is after $t - (p_k + \delta_k^I)$. At most $\lceil \frac{p_k + \delta_k^I + \Delta}{p_k} \rceil$ jobs of T_k can be released in $[t - p_k - \delta_k^I, t + \Delta)$. Hence, if T_k shares an object with T_i, then the number of interferences, and hence, failures, due to T_k in $[t, t + \Delta)$ is bounded from above by (number of jobs of T_k that can potentially execute in the interval) $\times$ (the number of retry loops of T_k that share an object with T_k). Summing over all the tasks that share objects with T_i, the claim follows. ■

Analysis for hard real-time. In a hard real-time system, every job J of T_i needs to complete executing within an interval of length p_i. Hence, the length Δ of the interval in which J can be interfered with is at most p_i. Also, $\delta_k = 0$ holds for every T_k. Thus, by Claim 1, the inflated execution cost of T_i due to interferences in retry loops, and hence, failed object accesses, is

$$e_i^{(s)} \leq e_i + \sum_{T_k \in \gamma_i} (\left\lceil \frac{p_i}{p_k} \right\rceil + 1) \cdot \beta_{i,k} \cdot r_{\max}. \qquad (11)$$

An EDF schedulability test in which $e_i^{(s)}$ is used instead of e_i will serve as a sufficient schedulability test for lock-free synchronization under EDF. (Recall that there is no blocking with lock-free.)

Analysis for soft real-time. In the absence of interferences, every job of T_i would complete executing within an interval of length $p_i + \delta_i$. However, accounting for interferences can lead to higher execution costs for the tasks, which in turn can lead to a higher tardiness bound. An increase in the tardiness bound amounts to an increase in the duration Δ of the interval in which each job can be interfered. Further, δ_k^I is unknown to begin with. Thus, bounding the number of interferences, and hence, tardiness for tasks in soft real-time systems is a bit complicated and has to be done iteratively. One set of iterative formulas is given below. (A superscript of j indicates that the value is computed in the jth iteration.) The computation of δ_i^j is by Theorem 1 and that of e_i^{j+1} is by Claim 1. Δ_i^j is an upper bound in iteration j to the length of the interval in which each job of T_i can be interfered.

$$
\begin{aligned}
e_i^1 &= e_i, \qquad u_i^j = \frac{e_i^j}{p_i} \\
\delta_i^j &= \frac{\sum_{i=1}^{\Lambda^j} \max(\epsilon_i^j, r_{\max}) - e_{\min}^j}{m - \sum_{i=1}^{\Lambda^j - 1} \mu_i^j} + e_i^j \\
\Delta_i^j &= p_i + \delta_i^j \\
e_i^{j+1} &= e_i + \sum_{T_k \in \gamma_i} \left(\left\lceil \frac{\Delta_i^j + \delta_k^j}{p_k} \right\rceil + 1 \right) \cdot \beta_{i,k} \cdot r_{\max}
\end{aligned}
$$

In iteration j, we assume that δ_k^I is at most δ_k^j and continue iterating until either **(i)** $e_i^{j+1} = e_i^j$ for all i, *i.e.*, when the tardiness bound converges for all the tasks, or **(ii)** $\sum_{i=1}^{n} u_i^j > m$, in which case bounded tardiness cannot be guaranteed on m processors.

5 Performance Evaluation

In this section, we present an evaluation of the two approaches.

As a first step, we determined typical execution times for common operations (such as insert, delete, lookup) on simple data structures guarded using queue locks. The execution times were measured on an SMP system with four 2.7 GHz Pentium 4 processors, 2 GB memory, 8 KB L1 data cache, and 512 KB unified L2 cache running Linux 2.6. The data structures considered were an eight-word buffer, a stack, a queue, a doubly-linked list, and a binary heap. (The list and heap were initialized with 50 nodes and grew to up to 1,000 and 30,000 nodes, respectively.) The maximum time required for any operation, including the time required to queue a lock request to the spin queue, but excluding the spinning time, was in the 1–5 μs range. Though lock-free implementations of the same operations using retry loops differ from their queue-lock based counterparts, by extrapolating from the above measurements, we concluded that execution costs of retry loops would also fall in the same range. Hence, in our simulations, we varied contention-free costs for object operations uniformly in the 1.3–6.5 μs range for both the approaches.

Next, sample data was generated as follows. Simulations were conducted for $m = 4$ and $m = 8$ processors. To reasonably constrain the experiments, task parameters were restricted as follows. The maximum number of tasks, N, in each task set was restricted to 20 when $m = 4$, and to 40 when $m = 8$. The maximum utilization of any task, $u_{\max}$, was systematically chosen from the set $\{0.1, 0.2, 0.3, 0.5\}$. Tasks were added to each task set until either the limit on the number of tasks or on the total system utilization was reached. The utilization of each task was uniformly distributed in the range $(0.0, u_{\max}]$.

For each task set, the maximum number of object operations per task, k, was chosen randomly between 1 and 10 and the total number of shared objects was fixed at $\frac{N \cdot k}{m/2}$. Thus, on an average, each object was shared among $m/2$ tasks. The actual number of object operations per task was uniformly distributed between 1 and k, where the cost of each operation in the absence of contention was chosen as specified above. Finally, the execution cost of each task, excluding that due to operations on shared objects, was uniformly distributed in the range $[50.0, 500.0]$ μs. There is not much guidance on how to assign execution costs. Our choice is based on the measured object-access costs reported above and the fact that the size of a quantum is typically at least 1ms on modern systems. Note that the higher the total execution cost, the lower the synchronization overhead would be.

For each task set generated, the inflation to task utilizations and the total system utilization were computed as described in Sec. 4 for both the queue-lock and lock-free approaches. Results are plotted in Fig. 1. The number of samples for each data point in the graphs was around 2,000 (except insets (e) and (f), and the curve for $m = 8$ and $u_{\max} = 0.5$ in inset (c), for which, as explained below, a large percentage of the samples was discarded). For hard real-time systems, overhead is reported in terms of the increase to total utilization, while for soft real-time systems, percentage increase to the tardiness bound is reported, in addition. For the comparison of tardiness bounds with and without synchronization to be meaningful, task sets whose total utilizations exceeded m upon inflation were discarded in computing increases to tardiness. Similarly, any task set with at least a task whose utilization exceeded 1.0 upon inflation was excluded in determining both the increase to utilization and tardiness.

Synchronization overheads for queue locks is shown in insets (a)–(c) of Fig. 1. Referring to insets (a) and (b), when $m = 4$, the increase to the total utilization with queue locks is at most 0.25 (6.25% of 4) for $k \leq 3$ and is at most 0.5 (12.5% of 4) for $k \leq 6$. For $m = 8$, the required inflation is at most 1.0 (12.5% of 8) when $k \leq 2$. In inset (a), the utilization loss for $u_{\max} = 0.5$ is lower than that for $u_{\max} = 0.2$ due to a decrease in the number of tasks. The trend in inset (b) is different due to a higher total utilization for the soft real-time case. Here the curves for $u_{\max} = 0.3$ and $u_{\max} = 0.5$ coincide. No task set was discarded with queue locks for the hard case, and when $k \leq 2$ or $u_{\max} < 0.3$ for the soft case. However, for the soft case, around 2% of the task sets were discarded for $u_{\max} \geq 0.3$ and $k = 2$. Inset (c) shows that for the task systems generated, the increase to tardiness is around 10% when $m = 4$ (resp., $m = 8$) and $k \leq 3$ (resp., $k \leq 2$), which are also reasonable. Also, the percentage of task sets discarded for these cases is minimal. Thus, when the number of object operations per task is at most two or three, *i.e.*, when the degree of sharing is reasonable (which we believe is the common case), queue locks provide an efficient synchronization solution. On the other hand, we believe that at higher degrees of sharing, even more complex synchronization approaches may not be capable of performing much better. This is because, as sharing increases, opportunities for concurrently executing the tasks decreases, leading to wasted processor time, to the point where execution is reduced to an almost sequential one at extreme degrees of sharing.

Insets (d) and (e) depict the inflation in utilization with the lock-free approach, for the hard and soft cases, respectively. As expected, the performance of this approach is poorer than queue locks. Though the plots in inset (e) seem comparable to, and in some cases better than, those in inset (d), in computing the overheads in inset (e), a high percentage of task sets had to be discarded as the inflated utilizations of tasks exceeded 1.0. For hard task systems, no task set was discarded and the inflation is reasonable for $m = 4$, if $k \leq 3$, and for $m = 8$, if k is at most one.

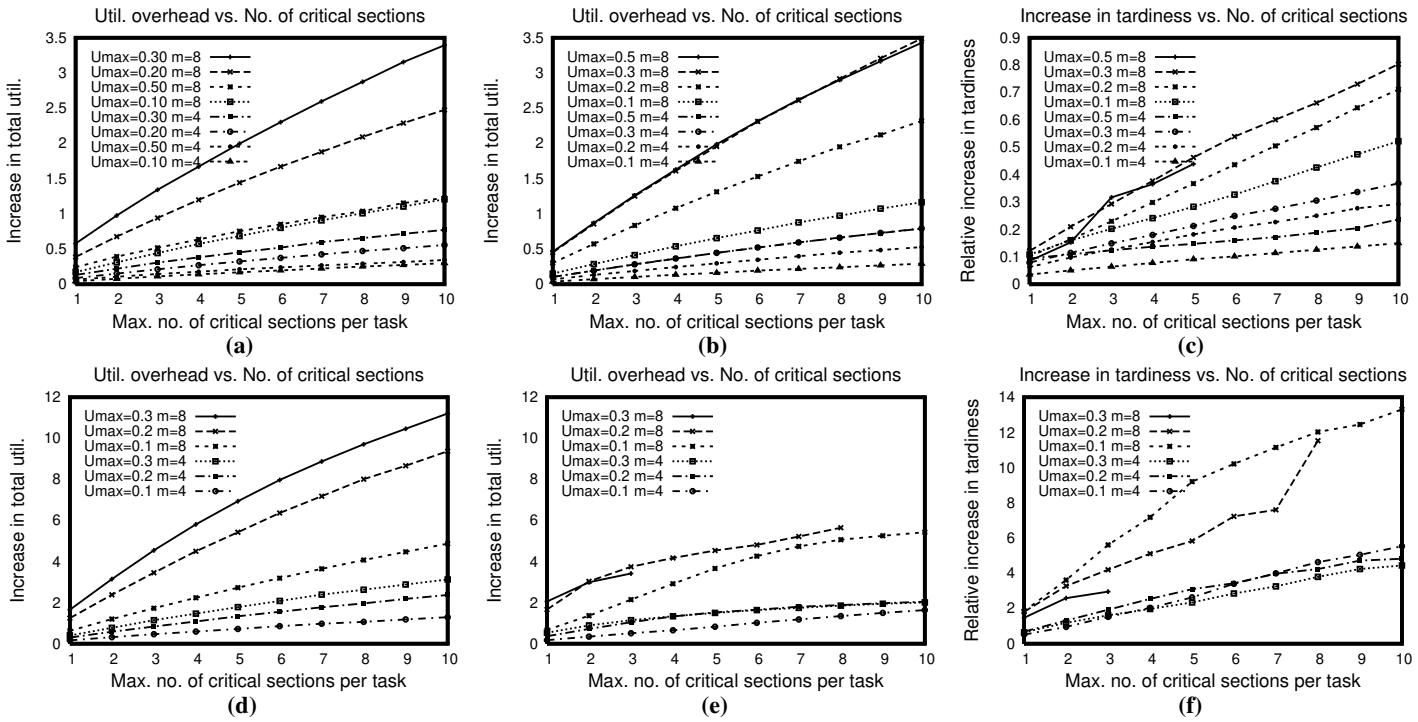

Figure 1: Synchronization overheads for queue locks and the lock-free approach. Inflation in total utilization for hard tasks with **(a)** queue locks and **(d)** lock-free, and for soft tasks with **(b)** queue locks and **(e)** lock-free. Relative increase in tardiness for **(c)** queue locks and **(f)** lock-free. (The order of the legends and curves coincide in all insets except in (e) and (f). Discrepancies here are due to the increase in the % of discarded task sets with increasing total utilization.)

However, for the soft case, the performance (especially, the percentage of undiscarded task sets) is poor except if $m = 4$, $k \leq 2$, and $u_{\max} \leq 0.2$. Even here, the increase in tardiness (refer inset (f)) is close to a 100%, which may not be acceptable. One reason for the poor performance may be the presence of tasks with extremely low utilizations. For instance, several task sets contain a few tasks with utilizations as low as $u_{\max}/100$. The number of interferences to such low-utilization tasks could potentially be in the hundreds, leading to high retry costs. We believe that the results would be much better if the ratio of the maximum to minimum utilization is restricted to at most 10.0. Another reason for the poor results is the pessimism in the analysis. However, improving the analysis does not seem to be simple, either. Finally, the disparity in the hard and soft real-time performance is partly due to the higher uninflated total utilization for the latter.

6 Conclusion

We have taken an initial step towards developing a generic resource-sharing framework for sporadic real-time tasks scheduled on a multiprocessor under global EDF. Based on evidence that suggests that a predominate use of synchronization constructs is for coordinating accesses to simple objects such as queues, stacks, and linked lists, we have proposed and evaluated two approaches for sharing such simple objects: one based on queue-based spin locks, and a second based on lock-free algorithms. Results of performance-evaluation studies show that queue locks impose very little overhead and may be very appropriate for both hard and soft real-time systems when the number of shared-object operations per task is small. Though such locks may not be appropriate for nested critical sections, in general, their use may be sufficient if the nesting is not deep and objects are acquired in order. The overhead of the lock-free approach is higher than that of queue locks; nevertheless, this approach may be reasonable in the absence of kernel support when the number of processors and the number of object calls are both low.

References

[1] J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. *ACM Transactions on Computer Systems*, 15(2):134–165, May 1997.

[2] T. P. Baker. Multiprocessor EDF and deadline monotonic schedulability analysis. In *Proceedings of the 24th IEEE Real-Time Systems Symposium*, pages 120–129, December 2003.

[3] S. Baruah, N. Cohen, C.G. Plaxton, and D. Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, June 1996.

[4] M. Bertogna, M. Cirinei, and G. Lipari. Improved schedulability analysis of EDF on multiprocessor platforms. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 209–218, July 2005.

[5] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. In Joseph Y. Leung, editor, *Handbook on Scheduling Algorithms, Methods, and Models*, pages 30.1–30.19. Chapman Hall/CRC, Boca Raton, Florida, 2004.

[6] U. Devi and J. Anderson. Tardiness bounds under global EDF scheduling on a multiprocessor. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, pages 330–341, December 2005.

[7] J. Goossens, S. Funk, and S. Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-Time Systems*, 25(2-3):187–205, 2003.

[8] P. Holman and J. Anderson. Locking in Pfair-scheduled multiprocessor systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, pages 149–158, December 2002.

[9] P. Holman and J. Anderson. Object sharing in Pfair-scheduled multiprocessor systems. In *Proceedings of the 14th Euromicro Conference on Real-Time Systems*, pages 111–120, June 2002.

[10] L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.

[11] J.W.S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[12] J. Mellor-Crummey and M. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Transactions on Computer Systems*, 9(1):21–65, February 1991.

[13] D. O'Hallaron. Spark98: Sparse matrix kernels for shared memory and message passing systems. Technical Report CMU-CS-97-178, Carnegie Mellon University, Oct. 1997.

[14] R. Rajkumar. Real-time synchronization protocols for shared memory multiprocessors. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 116–123, 1990.

[15] H. Takada and K. Sakamura. A novel approach to multiprogrammed multiprocessor synchronization for real-time kernels. In *Proceedings of the 18th IEEE Real-Time Systems Symposium*, pages 134–143, 1997.

[16] P. Tsigas and Y. Zhang. Evaluating the performance of non-blocking synchronization on shared-memory multiprocessors. In *Proceedings of the 2001 ACM SIGMETRICS Int'l Conference on Measurement and Modeling of Computer Systems*, pages 320–321. ACM Press, 2001.

[17] P. Tsigas and Y. Zhang. Integrating non-blocking synchronisation in parallel applications: performance advantages and methodologies. In *Proceedings of the Third Int'l Workshop on Software and Performance*, pages 55–67. ACM Press, 2002.

[18] P. Valente and G. Lipari. An upper bound to the lateness of soft real-time tasks scheduled by EDF on multiprocessors. In *Proceedings of the 26th IEEE Real-Time Systems Symposium*, pages 311–320, December 2005.

[19] S. Woo, M. Ohara, E. Torrie, J. Singh, and A. Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd Int'l Symposium on Computer Architectures*, pages 24–36, 1995.

[20] V. Yadaiken. Against priority inheritance. Technical report, Finite State Machine Labs, June 2002.

Appendix: Proof of Lemma 2

In this appendix, we give a proof of Lemma 2. For conciseness, we omit specifying the schedule $\mathcal{S}$ (the final argument) in functions LAG and B.

In [6], we showed that if a task does not execute continuously within a non-busy interval in an EDF schedule, then its lag at the end of the interval is at most zero. This property holds for a non-blocking, non-busy interval of an EDF-hybrid schedule also and is stated formally below.

Lemma 4 (from [6]) *Let $[t, t')$ be a maximally non-blocking, non-busy interval in $[0, t_d)$ in $\mathcal{S}$ and let T_k be a task in τ with a job in Ψ that is active or pending at t'^-. If T_k does not execute continuously in $[t, t')$, then $\mathsf{lag}(T_k, t') \leq 0$.*

To see why this lemma holds, note that, because $[t, t')$ is maximally non-busy and is non-blocking, at every instant in this interval, at least one processor is idle or a job in $\overline{\Psi}$ is executing while no job in Ψ is waiting. The absolute deadline of a job in $\overline{\Psi}$ is after t_d. Hence, if T_k is not executing at t'^-, then it has no pending work at t'^-, and hence, its lag at t' is at most zero. On the other hand, if T_k is executing at t'^-, but was not executing some time earlier in $[t, t')$, then it must have had no pending work when its most-recent job was released and must have executed continuously since then. In this case too, its lag cannot exceed zero.

The lemma that follows bounds the lag of a task at any arbitrary time at or before t_d.

Lemma 5 $\mathsf{lag}(T_k, t, \mathcal{S}) \leq x \cdot u_k + e_k$ *holds for every task T_k for any time instant $t \leq t_d$.*

Proof: If no job of T_k is pending at t, then T_k's lag at t is at most zero and the lemma holds trivially. Hence, assume that one or more jobs of T_k are pending at t and let $T_{k,q}$ be the earliest pending job. Let $\delta_{k,q} < e_k$ be the amount of time that $T_{k,q}$ executed for before t. We prove the lemma for the case $d_{k,q} < t$, leaving the case $d_{k,q} \geq t$ to the reader. The amount of work pending for $T_{k,q}$ at t is $e_k - \delta_{k,q}$. T_k is allocated at most u_k time at every instant after $d_{k,q}$ in PS_τ. Therefore, $\mathsf{lag}(T_k, t, \mathcal{S}) \leq (t - d_{k,q}) \cdot u_k + e_k - \delta_{k,q}$ holds. By (P), the tardiness of $T_{k,q}$ is at most $x + e_k$. Therefore, $t + e_k - \delta_{k,q} \leq d_{k,q} + x + e_k$, *i.e.*, $t - d_{k,q} \leq x + \delta_{k,q}$ holds. Substituting for $t - d_{k,q}$ in the expression for lag, we arrive at the lemma. ■

We next make the following three claims, which will be used in proving later lemmas.

Claim 2 *Let $[t, t')$ be a maximally blocking, non-busy interval in $[0, t_d)$ in $\mathcal{S}$. Then, the following hold.* **(i)** $t > 0$. **(ii)** *Any job that is in $\overline{\Psi}$ and is executing at $\hat{t}$, where $t \leq \hat{t} < t'$, executes a single non-preemptive section continuously in $[t^-, \hat{t}]$.*

Proof: Since every job of Ψ has an earlier deadline than a job in $\overline{\Psi}$, a job in Ψ cannot be blocked at time 0 by a job in $\overline{\Psi}$. Therefore $t > 0$ holds. By the definition of $[t, t')$, no job of Ψ (including jobs that are blocked at t) is blocked by a job in $\overline{\Psi}$ at t^-. Hence, it cannot be the case that a job in Ψ is blocked at t due to a job in $\overline{\Psi}$ commencing executing a non-preemptive section at t. Rather, the blocking non-preemptive section should have commenced execution before t and the blocked job becomes ready at t (because it is either released or preempted by a higher-priority job that is in Ψ at t). Similarly, since every instant in $[t, t')$ is a blocking instant at which one or more ready jobs of Ψ are waiting, no job in $\overline{\Psi}$ can be executing in a preemptive section and no non-preemptive section of a job in $\overline{\Psi}$ can commence execution in (t, t'). The claim follows from these facts. ■

Claim 3 *Let $[t, t')$ be a maximally blocking, non-busy interval in $[0, t_d)$ in $\mathcal{S}$ such that $\mathsf{LAG}(\Psi, t') > \mathsf{LAG}(\Psi, t)$. Then, at t and t^-, at most Λ tasks have their jobs in Ψ executing.* (*Note: The tasks executing at t and t^- need not be the same.*)

Proof: Because $\mathsf{LAG}(\Psi, t') > \mathsf{LAG}(\Psi, t)$ holds, by Lemma 1 and (1), there exists at least one time instant $\hat{t}$, where $t \leq \hat{t} < t$,

such that at most Λ tasks have executing jobs in Ψ at $\hat{t}$. Let $k \leq \Lambda$ denote the number of such tasks. Then, since $[t, t')$ is a blocking, non-busy interval, no processor is idle in the interval. Hence, exactly $m - k$ jobs from $\overline{\Psi}$ are executing at $\hat{t}$. By Claim 2, $t > 0$ and each of the $m - k$ jobs is executing continuously in $[t^-, \hat{t}]$. Hence, at t^- and t, at most $k \leq \Lambda$ tasks may have executing jobs in Ψ. ■

Claim 4 *For any* $k \leq \Lambda$, $(\sum_{i=1}^{k} x \cdot \mu_i + \epsilon_i) + (m - k) \cdot b_{\max} \leq (\sum_{i=1}^{\Lambda} x \cdot \mu_i + \max(\epsilon_i, b_{\max})) + (m - \Lambda) \cdot b_{\max}$.

Proof: The claim trivially holds for $k = \Lambda$. Below, we prove for $k < \Lambda$.

$$
\begin{aligned}
&(\sum_{i=1}^{k} x \cdot \mu_i + \epsilon_i) + (m - k) \cdot b_{\max} \\
&= (\sum_{i=1}^{k} x \cdot \mu_i + \epsilon_i) + (m - \Lambda) \cdot b_{\max} + (\Lambda - k) \cdot b_{\max} \\
&\leq (\sum_{i=1}^{k} x \cdot \mu_i + \max(\epsilon_i, b_{\max})) + (m - \Lambda) \cdot b_{\max} \\
&\quad + \sum_{i=k+1}^{\Lambda} \max(\epsilon_i, b_{\max}) \\
&\leq (\sum_{i=1}^{\Lambda} x \cdot \mu_i + \max(\epsilon_i, b_{\max})) + (m - \Lambda) \cdot b_{\max}
\end{aligned}
$$

■

The next two lemmas show how to bound LAG at the end of a non-blocking, non-busy interval and a blocking, non-busy interval.

Lemma 6 *Let* $[t, t')$ *be a maximally non-blocking, non-busy interval in* $[0, t_d)$ *in* $\mathcal{S}$ *such that either* $\mathsf{LAG}(\Psi, t') > \mathsf{LAG}(\Psi, t)$ *or at most* Λ *tasks are executing at* t'^-. *Let* k *denote the number of tasks that are executing jobs in* Ψ *continuously in* $[t, t')$. *Then,* **(i)** $k \leq \Lambda$, **(ii)** $\mathsf{LAG}(\Psi, t') \leq \sum_{i=1}^{\Lambda}(x \cdot \mu_i + \epsilon_i)$, *and* **(iii)** $\mathsf{LAG}(\Psi, t') + \mathsf{B}(\Psi, t') \leq (\sum_{i=1}^{\Lambda} x \cdot \mu_i + \max(\epsilon_i, b_{\max})) + (m - \Lambda) \cdot b_{\max}$.

Proof: Let α denote the subset of all tasks in τ that are executing jobs in Ψ continuously in $[t, t')$. Then, $|\alpha| = k$ holds. If at most Λ tasks are executing at t'^-, then clearly $k \leq \Lambda$ holds. On the other hand, if $\mathsf{LAG}(\Psi, t') > \mathsf{LAG}(\Psi, t)$ holds, then, by Lemma 1,

$$|\alpha| = k < \max_{t \leq \hat{t} < t'} \{U_{sum}(\Psi, \hat{t})\} \leq U_{sum}(\tau). \tag{12}$$

Because k is an integer, by (12) and (1), we have $k \leq \Lambda$, which establishes Part (i).

By (6), the LAG of Ψ at t' is at most the sum of the lags at t' of all tasks in τ with at least one job in Ψ that is active or pending at t'^-. By Lemma 4, the lag of such a task that does not execute continuously in $[t, t')$ is at most zero. Hence, to determine an upper bound on LAG at t', it is sufficient to determine an upper bound on the sum of lags of such tasks that are executing continuously in $[t, t')$, *i.e.*, tasks in α . Thus,

$$
\begin{aligned}
\mathsf{LAG}(\Psi, t') &\leq \sum_{T_i \in \alpha} \mathsf{lag}(T_i, t', \mathcal{S}) \\
&\leq \sum_{T_i \in \alpha} (x \cdot u_i + e_i) \qquad \{\text{by Lemma 5}\} \\
&\leq \sum_{i=1}^{k} (x \cdot \mu_i + \epsilon_i). \qquad (13) \\
&\qquad \{\text{by (12) and the definintions of } \mu_i \text{ and } \epsilon_i\}
\end{aligned}
$$

Hence, Part (ii) follows, because $k \leq \Lambda$ (from Part (i)). Finally, since at least k jobs in Ψ are executing at t'^-, at most $m - k$ jobs of $\overline{\Psi}$ can be executing at t'^-. The maximum time that all such jobs in $\overline{\Psi}$ can execute for after t in non-preemptive sections that commenced before t, *i.e.*, $\mathsf{B}(\tau, \Psi, t')$ is at most $(m - k) \cdot b_{\max}$. Hence, by (13), $\mathsf{LAG}(\Psi, t') + \mathsf{B}(\tau, \Psi, t') \leq \sum_{i=1}^{k}(x \cdot \mu_i + \epsilon_i) + (m - k) \cdot b_{\max} \leq \sum_{i=1}^{\Lambda}(x \cdot \mu_i + \max(\epsilon_i, b_{\max})) + (m - \Lambda) \cdot b_{\max}$, where the last inequality follows from Claim 4. ■

Lemma 7 *Let* $[t, t')$ *be a maximally blocking, non-busy interval in* $[0, t_d)$ *in* $\mathcal{S}$ *such that* $k \leq \Lambda$ *tasks have their jobs in* Ψ *executing at* t. *Then, we have the following.* **(i)** $\mathsf{LAG}(\Psi, t') \leq (\sum_{i=1}^{\Lambda} x \cdot \mu_i + \epsilon_i) + b_{\max}$, *and* **(ii)** $\mathsf{LAG}(\Psi, t') + \mathsf{B}(\tau, \Psi, t') < (\sum_{i=1}^{\Lambda} x \cdot \mu_i + \max(\epsilon_i, b_{\max})) + (m - \Lambda) \cdot b_{\max}$.

Proof: Let $\mathcal{J}$ denote the set of all jobs of $\overline{\Psi}$ that are executing at t, and hence are blocking one or more jobs of Ψ. Let $b = |\mathcal{J}|$.Since k jobs from Ψ are executing at t and $[t, t')$ is a blocking, non-busy interval (hence, no processor is idle), we have

$$|\mathcal{J}| = b = m - k. \tag{14}$$

By our definition of t and Claim 2, it follows that t^- is a non-blocking, non-busy instant. By (14) and Claim 2 again, it follows that at least $m - k$ jobs of $\overline{\Psi}$ are executing at t^-. Therefore, at most k jobs from Ψ can be executing at t^-. Since no job of Ψ that is not executing at a non-blocking, non-busy instant can be pending (and hence no such job may have a positive lag at t), by (6), we have the following.

$$
\begin{aligned}
&\mathsf{LAG}(\Psi, t) \\
&\leq \sum_{\substack{\{T_i \in \tau \,:\, T_{i,j} \text{ is in } \Psi, \text{ and is} \\ \text{executing at } t^-\}}} \mathsf{lag}(T_i, t, \mathcal{S}) \\
&\leq \sum_{\substack{\{T_i \in \tau \,:\, T_{i,j} \text{ is in } \Psi, \text{ and is} \\ \text{executing at } t^-\}}} x \cdot u_i + e_i \qquad \{\text{by Lemma 5}\} \\
&\leq \sum_{i=1}^{k} (x \cdot \mu_i + \epsilon_i) \quad \{\text{because, as discussed above, at most } k \\
&\qquad \text{tasks with jobs in } \Psi \text{ are executing at } t^-\} \qquad (15)
\end{aligned}
$$

By (8), in PS_τ, the total allocation to Ψ in $[t, t')$ cannot exceed $(t' - t) \cdot U_{sum}(\tau)$. By (14) and Part (ii) of Claim 2, at most $m - k$ jobs of $\overline{\Psi}$ execute at any instant in $[t, t')$. Hence, because $[t, t')$ is a maximally blocking interval, at least k jobs from Ψ execute at each instant in $[t, t')$ in $\mathcal{S}$. Therefore, the total allocation to Ψ in $\mathcal{S}$ is at least $k \cdot (t' - t)$. Hence, by (9), we have

$$
\begin{aligned}
&\mathsf{LAG}(\Psi, t') \\
&\leq (t' - t)(U_{sum}(\tau) - k) + \mathsf{LAG}(\Psi, t)
\end{aligned}
$$

$$
\begin{aligned}
&\leq (t'-t)(\Lambda+1-k)+\sum_{i=1}^{k}(x\cdot\mu_i+\epsilon_i) \quad \{\text{by (1) and (15)}\}\\
&\leq b_{\max}\cdot(\Lambda+1-k)+\sum_{i=1}^{k}(x\cdot\mu_i+\epsilon_i)\\
&\qquad \{\text{because, by Claim 2, every job of } \mathcal{J} \text{ executes a single non-preemptive section in } [t,t'), \text{ and hence, } t'-t<b_{\max}\}\\
&= b_{\max}\cdot(\Lambda-k)+b_{\max}+\sum_{i=1}^{k}(x\cdot\mu_i+\epsilon_i)\\
&\leq b_{\max}\cdot(\Lambda-k)+b_{\max}\\
&\quad +\sum_{i=1}^{\Lambda}x\cdot\mu_i+\sum_{i=1}^{k}\epsilon_i \qquad \{\text{because } k\leq\Lambda\}\\
&\leq \sum_{i=1}^{\Lambda}(x\cdot\mu_i+\max(\epsilon_i,b_{\max}))+b_{\max}. \quad \{\text{because } k\leq\Lambda\}
\end{aligned}
$$

That establishes Part (i). Next, we determine an upper bound on the sum of $\mathsf{LAG}(\Psi,t')$ and $\mathsf{B}(\tau,\Psi,t')$. For this, we first determine an upper bound on $\mathsf{B}(\tau,\Psi,t)$. By Claim 2, every job of $\overline{\Psi}$ that is executing anywhere in $[t,t')$ is in $\mathcal{J}$. Furthermore, every such job executes a single non-preemptive section in $[t^-,t')$. Hence, for any time u in $[t,t')$, $\mathsf{B}(\tau,\Psi,u)$ is given by the amount of work pending at u for the non-preemptive sections executing at t of jobs in $\mathcal{J}$. Let J be a job in $\mathcal{J}$. Then, the amount of work that can be pending for its non-preemptive section executing at t is at most $b_{\max}$. Therefore, by (14), we have $\mathsf{B}(\tau,\Psi,t)\leq(m-k)\cdot b_{\max}$, and hence, by (15), we have

$$
\begin{aligned}
&\mathsf{LAG}(\Psi,t)+\mathsf{B}(\tau,\Psi,t)\\
&< (\sum_{i=1}^{k}x\cdot\mu_i+\epsilon_i)+(m-k)\cdot b_{\max}\\
&\leq (\sum_{i=1}^{\Lambda}x\cdot\mu_i+\max(\epsilon_i,b_{\max}))+(m-\Lambda)\cdot b_{\max}, \qquad (16)
\end{aligned}
$$

where the last inequality follows from Claim 4.

Finally, we are left with determining an upper bound on the sum of LAG and B at t'. Let $X\leq\mathsf{B}(\tau,\Psi,t)$ denote the total amount of time that jobs in $\mathcal{J}$ execute on all m processors in $[t,t')$. (For example, if there are two jobs in $\mathcal{J}$, with one job executing for the entire interval and the second executing for the first half of the interval, then $X=3(t'-t)/2$.) Because $[t,t')$ is maximally blocking, no processor is idle in $[t,t')$. Hence, the total time allocated to jobs in Ψ in $[t,t')$, $\mathsf{A}(\mathcal{S},\Psi,t,t')$, is equal to $m\cdot(t'-t)-X$. In PS_τ, jobs in Ψ could execute for at most $U_{sum}(\tau)\cdot(t'-t)$ time, *i.e.*, $\mathsf{A}(\mathsf{PS}_\tau,\Psi,t,t')\leq U_{sum}(\tau)\cdot(t'-t)$. Therefore, $\mathsf{LAG}(\Psi,t')=\mathsf{LAG}(\Psi,t)+\mathsf{A}(\mathsf{PS}_\tau,\Psi,t,t')-\mathsf{A}(\mathcal{S},\Psi,t,t')\leq\mathsf{LAG}(\Psi,t)+(U_{sum}(\tau)-m)\cdot(t'-t)+X\leq\mathsf{LAG}(\Psi,t)+X$. However, since jobs in $\mathcal{J}$ execute for a total time of X in $[t,t')$, the pending work for non-preemptive sections of jobs in $\mathcal{J}$, and hence, those in $\mathcal{B}(\tau,\Psi,t')$ at t', *i.e.*, $\mathsf{B}(\tau,\Psi,t')$, is at most $\mathsf{B}(\tau,\Psi,t)-X$. Thus, $\mathsf{LAG}(\Psi,t')+\mathsf{B}(\tau,\Psi,t')\leq\mathsf{LAG}(\Psi,t)+\mathsf{B}(\tau,\Psi,t)$, which by (16), establishes Part (ii) of the lemma. ■

Finally, Lemmas 6 and 7 can be used to establish the following.

Lemma 2 $\mathsf{LAG}(\Psi,t_d)+\mathsf{B}(\tau,\Psi,t_d)\leq\sum_{i=1}^{\Lambda}(x\cdot\mu_i+\max(\epsilon_i,b_{\max}))+(m-\Lambda)\cdot b_{\max}$.

Proof: We consider the following three cases based on the nature of t_d^-.

Case 1: t_d^- is a busy instant. In this case, because t_d^- is busy, by definition, $\mathsf{B}(\tau,\Psi,t_d)=0$. By Lemma 1, the LAG of Ψ can increase only across a non-busy interval. Therefore, LAG at t_d is at most that at the end of the latest non-busy instant before t_d. If no non-busy interval exists in $[0,t_d)$, then $\mathsf{LAG}(\Psi,t_d)\leq\mathsf{LAG}(\Psi,0)=0$. Otherwise, by Part (ii) of Lemma 6 and Part (i) of Lemma 7 is at most $\sum_{i=1}^{\Lambda}(x\cdot\mu_i+\max(\epsilon_i,b_{\max}))+b_{\max}$. By (1), $m-\Lambda\geq1$, and hence, the lemma follows.

Case 2: t_d^- is a blocking, non-busy instant. Let $t<t_d$ be the earliest instant before t_d such that $[t,t_d)$ is a maximally blocking, non-busy interval. Let k denote the number of tasks whose jobs in Ψ are executing at t. We consider the following two subcases.

Subcase 2(a): $\mathsf{LAG}(\Psi,t_d)>\mathsf{LAG}(\Psi,t)$ or $k\leq\Lambda$. If $\mathsf{LAG}(\Psi,t_d)>\mathsf{LAG}(\Psi,t)$ holds, then by Claim 3, $k\leq\Lambda$ holds. Therefore, for this subcase, the lemma follows from Part (ii) of Lemma 7.

Subcase 2(b): $k>\Lambda$ and $\mathsf{LAG}(\Psi,t_d)\leq\mathsf{LAG}(\Psi,t)$. By the conditions of this subcase, it follows that LAG at t_d is bounded from above by the LAG at the end of the latest non-blocking or blocking non-busy interval before t across which LAG increases. If no such interval exists, then $\mathsf{LAG}(\tau,t_d)\leq\mathsf{LAG}(\tau,0)=0$. Otherwise, by Part (ii) of Lemma 6 and Part (i) of Lemma 7, we have

$$
\mathsf{LAG}(\Psi,t_d)\leq(\sum_{i=1}^{\Lambda}x\cdot\mu_i+\epsilon_i)+b_{\max}. \qquad (17)
$$

Since $k>\Lambda$ holds, at most $m-\Lambda-1$ jobs from $\overline{\Psi}$ can be executing at t and by Claim 2, at most $m-\Lambda-1$ such jobs can be executing at t_d^- as well. The amount of time such jobs can execute past t_d in non-preemptive sections is at most $b_{\max}$. Thus, $\mathsf{B}(\tau,\Psi,t_d)\leq(m-\Lambda-1)\cdot b_{\max}$ holds. Therefore, by (17), $\mathsf{LAG}(\Psi,t_d)+\mathsf{B}(\tau,\Psi,t_d)<(\sum_{i=1}^{\Lambda}x\cdot\mu_i+\max(\epsilon_i,b_{\max}))+(m-\Lambda)\cdot b_{\max}$ holds.

Case 3: t_d^- is a non-blocking, non-busy instant. The arguments for this case are somewhat similar to that used in Case 2. Let t denote the earliest time instant before t_d such that $[t,t_d)$ is a maximally non-blocking, non-busy interval. If either at most Λ tasks execute their jobs in Ψ at t_d^- or $\mathsf{LAG}(\Psi,t_d)>\mathsf{LAG}(\Psi,t)$, then the lemma holds for this case by Part (iii) of Lemma 6. The remaining possibility is that at least $\Lambda+1$ tasks execute their jobs in Ψ at t_d^- and $\mathsf{LAG}(\Psi,t_d)\leq\mathsf{LAG}(\Psi,t)$ holds. In this case, LAG at t_d is given by the LAG at the end of the latest non-busy interval before t across which LAG increases. If no such interval exists, then $\mathsf{LAG}(\Psi,t_d)\leq\mathsf{LAG}(\Psi,0)=0$. Otherwise, by Part (ii) of Lemma 6, and Claim 3 and Part (i) of Lemma 7, the LAG at the end of the latest non-busy interval before t across which LAG increases, and hence, $\mathsf{LAG}(\Psi,t_d)$, is at

most $(\sum_{i=1}^{\Lambda} x \cdot \mu_i + \epsilon_i) + b_{\max}$. Because at least $\Lambda + 1$ tasks execute their jobs in Ψ at t_d^- at most $m - \Lambda - 1$ jobs from $\overline{\Psi}$ can be executing at t_d^-. The amount of time each such job can execute past t_d in a non-preemptive section that commenced before t_d is at most $b_{\max}$. Thus, $\mathsf{B}(\tau, \Psi, t_d) \leq (m - \Lambda - 1) \cdot b_{\max}$, and hence, $\mathsf{LAG}(\Psi, t_d) + \mathsf{B}(\tau, \Psi, t_d) \leq (\sum_{i=1}^{\Lambda} x \cdot \mu_i + \max(\epsilon_i, b_{\max})) + (m - \Lambda) \cdot b_{\max}$ holds. ■