Synchronization for an optimal real-time scheduling algorithm on multiprocessors

Hyeonjoong Cho
ETRI
Daejoen, South Korea
Email: raycho@etri.re.kr

Binoy Ravindran
ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
Email: binoy@vt.edu

E. Douglas Jensen
The MITRE Corporation
Bedford, MA 01730-1420, USA
Email: jensen@mitre.org

***Abstract*— We consider several object sharing synchronization mechanisms including lock-based, lock-free, and wait-free sharing for LNREF [1], an optimal real-time scheduling algorithm on multiprocessors. We derive LNREF's minimum-required space cost for wait-free synchronization using the space-optimal wait-free algorithm. We then establish the feasibility conditions for lock-free and lock-based sharing under LNREF, and the concomitant tradeoffs. While the tradeoff between wait-free versus the other sharing is obvious, i.e., space and time costs, we show that the tradeoff between lock-free and lock-based sharing for LNREF hinges on the cost of the lock-free retry, blocking time under lock-based. Finally, we numerically evaluate lock-free and lock-based sharing for LNREF.**

I. Introduction

Multiprocessor architectures are gaining more interest because major processor manufacturers (Intel, AMD) are making them decreasingly expensive. Responding these trends, RTOS vendors are increasingly providing multiprocessor platform support. Responding to this trend, RTOS vendors are increasingly providing multiprocessor platform support — e.g., QNX Neutrino is now available for a variety of SMP chips [2]. But this exposes the critical need for real-time scheduling for multiprocessors — a comparatively undeveloped area of real-time scheduling which has recently received significant research attention, but is not yet well supported by the RTOS products. Consequently, the impact of cost-effective multiprocessor platforms for embedded systems remains nascent.

The Pfair class of algorithms [3] have been shown to be theoretically optimal—i.e., they achieve a schedulable utilization bound (below which all tasks meet their deadlines) that equals the total capacity of all processors. However, Pfair algorithms incur significant run-time overhead due to their quantum-based scheduling approach [4], [5]: under Pfair, tasks can be decomposed into several small uniform segments, which are then scheduled, causing frequent scheduling and migration.

We have presented another optimal real-time scheduling algorithm for multiprocessors, which is not based on time quanta in [1]. The algorithm called LNREF, is based on the fluid scheduling model and the fairness notion. However, we have assumed that all running tasks are independent each other and the object sharing synchronization for LNREF has not been considered yet.

We consider object sharing synchronization mechanisms for an optimal scheduling algorithm, LNREF, in this paper. Compared to the research efforts on multiprocessor scheduling, synchronization for multiprocessor scheduling has been less studied. For example, the synchronization under global EDF was considered only recently in [6]. We consider several resource sharing methods under multiprocessor scheduling, including wait-free, lock-free and lock-based for LNREF algorithms.

Most embedded real-time systems involve concurrent access to shared data objects, resulting in contention for those objects. Resolution of the contention directly affects the system's timeliness, and thus the system's behavior. Mechanisms that resolve such contention can be broadly classified into: (1) lock-based—e.g., Priority Inheritance and Ceiling protocols [7], Stack Resource Policy [8], DASA [9]; (2) wait-free—e.g., NBW protocol [10], Chen's protocol [11], [12], [13]; and (3) lock-free—e.g., [14].

Lock-based object sharing traditionally relies on mutual exclusion to obtain atomicity. Sometimes, its mutual exclusion incurs several disadvantages such as serialized access to shared objects, resulting in reduced concurrency and thus reduced resource utilization [14].

These drawbacks have motivated research on wait-free and lock-free object sharing in real-time systems. An implementation of wait-free protocols uses *multiple buffers* (e.g., a circular buffer) for writers and readers [10], [13], [15]. For the *single-writer/multiple-reader problem* (SWMR), wait-free buffers typically use multiple internal buffers for the shared object, where the number of internal buffers used is proportional to the maximum number of times the readers can be preempted by the writer, when the readers are reading. The maximum number of preemptions of a reader bounds the number of times the writer can update the object while the reader is reading. Thus, by using as many internal buffers as the worst-case number of preemptions of readers, the readers and the writer can continuously read and write in different buffers, respectively, and avoid interference.

On the other hand, lock-free protocols allow readers to concurrently read while the writer is writing (without acquiring locks), but the readers check whether their reading was interfered by the writer. If so, they read again. Thus, a reader continuously reads, checks, and retries until its read becomes successful. Since a reader's worst-case number of retries depends upon the worst-case number of times the reader is preempted by the writer, the additional execution-time overhead incurred for the retries is bounded by the number of preemptions.

Both wait-free and lock-free protocols incur additional costs with respect to their lock-based counterparts. Wait-free protocols generally incur additional space costs due to

their multiple internal buffer usage, which is infeasible in many small-memory, embedded real-time systems. Lock-free protocols generally incur additional time costs due to their retries, which is antagonistic to timeliness optimization.

We consider these object sharing methods for LNREF on multiprocessors. Similar to [6], we focus on the common case of concurrent, non-nested, serialized access to simple shared objects. Simple objects (e.g., buffers, queues, stacks) are the predominant shared objects in many embedded applications. For example, Tsigas and Yang observe that almost all synchronization in the SPLASH-2 and the Spark98 kernel benchmark suites are for simple objects [16], [17]. Thus, we focus on simple object sharing under LNREF.

For the LNREF algorithm, it is problematic to impose time costs for executing tasks. This is because, LNREF abides by the fairness notion that requires for each task to run at a certain rate in time intervals. (Note that LNREF does not maintain P-fairness [3].) Therefore, any prolonged execution of tasks may violate fairness. In this sense, it is appropriate to consider wait-free synchronization for LNREF, which does not impose any time costs, but incurs space cost. However, the space costs can be minimized using the optimal wait-free synchronization algorithm [15]. Thus, we derive the minimal required space costs for LNREF using the wait-free algorithm for the SWMR problem.

Despite wait-free synchronization's compatibility with the fairness notion, it is restricted in its use, e.g, overload case, as opposed to lock-free and lock-based sharing. Lock-free objects are known to work well particularly for simple objects like buffers, queues, and lists. In multiprocessor real-time systems, lock-free algorithms have been viewed as impractical, because deducing bounds on retries due to interferences across processors is difficult [18]. Holman *et al.* have bounded retries under Pfair scheduling by exploiting its tight synchrony, and under the observation that lock-free operations take very small time [18]. However, bounding retries is more difficult for asynchronous real-time scheduling, which does not depend on time quanta, e.g., LNREF. Thus, we consider an auxiliary method to bound retires, i.e., *non-preemptive area* (or NPA) surrounding each lock-free operation, inspired by [6]. If there is a scheduling event during the NPA and another task tries to preempt the processor, it will be blocked up to the end of the NPA. Thus, the NPA guarantees finite number of retries for lock-free object sharing. This allows us to use lock-free object sharing even during overloads as opposed to wait-free object sharing. In this paper, we present feasible conditions for LNREF with NPA-assisted lock-free object sharing.

Although lock-free is efficient for simple objects, lock-based object sharing is still needed to implement more complicated objects. We consider *queue-based spin locks* (or queue locks) for both LNREF. In queue locks [6], a task waits by busy-waiting, or spinning, on "spin variable" (i.e., repeatedly testing its status), and waiting tasks are ordered within a "spin queue". When a task attempts to acquire a lock, it appends its lock request onto the end of the spin queue. 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.

Similar to [6], we assume that shared object calls are implemented as critical sections accessed via queue locks invoked within non-preemptive regions. It is because if a task is preempted within a critical section, then the waiting times for any tasks that it blocks could significantly increase. Thus, in this paper, we consider shared objects implemented with queue-based spin locks within NPA for LNREF. As claimed in [6], the idea of the NPA is in line with views expressed by others. The founder of RTLinux recommends accessing simple critical sections non-preemptively in [19], for example. We then derive feasible conditions for LNREF under this lock-based scheme.

The rest of the paper is organized as follows: Section II shows an overview of LNREF. Section III discusses wait-free, lock-free, lock-based sharing for LNREF and then we compare the different schemes and identify tradeoffs. In Section IV, we numerically evaluate lock-free and lock-based schemes. We conclude the paper in Section V.

II. LNREF Scheduling Algorithm

LNREF [1] can meet all deadlines as long as the total utilization demand does not exceed the total processing capacity of all processors. In this section, we briefly introduce LNREF algorithm.

A. Model

We consider global scheduling, where task migration is not restricted, on an SMP system with M identical processors. We consider the application to consist of a set of tasks and tasks are assumed to arrive periodically at their release times. Each task has an execution time, and a relative deadline which is the same as its period. The utilization u_i of a task is defined as its execution time over its relative deadline and is assumed to be less than 1. Similar to [4], [20], we assume that tasks may be preempted at any time, and are independent, i.e., they do not share resources or have any precedences.

B. Time and Nodal Execution Time Plane

In the *fluid* scheduling model, each task executes at a constant rate at all times [21]. The quantum-based Pfair scheduling algorithm is based on the fluid scheduling model, as the algorithm constantly tracks the allocated task execution rate through task utilization. The Pfair algorithm's success in constructing optimal multiprocessor schedules can be attributed to *fairness* — informally, all tasks receive a share of the processor time, and thus are able to simultaneously make progress. P-fairness is a strong notion of fairness, which ensures that at any instant, no application is more than one quantum away from its due share (or fluid schedule) [3], [22].

Toward designing an optimal scheduling algorithm, we thus consider the fluid scheduling model and the fairness notion. To avoid Pfair's quantum-based approach, we consider an abstraction called the *Time and Nodal Execution Time Domain Plane* (or abbreviated as the T-N plane), where tokens representing

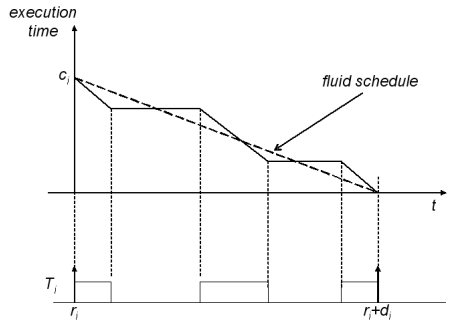

Fig. 1. Fluid Schedule versus a Practical Schedule

tasks move over time. We use the T-N plane to describe fluid schedules, and present a new scheduling algorithm that is able to track the fluid schedule without using time quanta.

Figure 1 illustrates the fundamental idea behind the T-N plane. For a task T_i with its release time r_i, execution time c_i and deadline d_i, the figure shows a 2-dimensional plane with time represented on the x-axis and the task's remaining execution time represented on the y-axis. If r_i is assumed as the origin, the dotted line from $(0, c_i)$ to $(d_i, 0)$ indicates the fluid schedule, the slope of which is utilization $-u_i$. Since the fluid schedule is ideal but practically impossible, the fairness of a scheduling algorithm depends on how much the algorithm approximates the fluid schedule path.

When T_i runs like in Figure 1, for example, its execution can be represented as a broken line between $(0, c_i)$ and $(d_i, 0)$. Note that task execution is represented as a line whose slope is -1 since x and y axes are in the same scale, and the non-execution over time is represented as a line whose slope is zero.

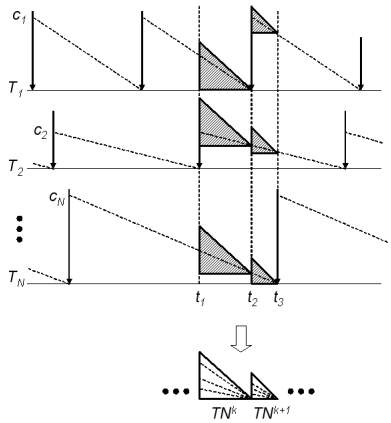

Fig. 2. T-N Planes

When N number of tasks are considered, their fluid schedules can be constructed as shown in Figure 2, and a right isosceles triangle for all tasks is found between every two consecutive scheduling events. We call this as the T-N plane TN^k, where k is simply increasing over time. The size of TN^k may change over k. The bottom side of the triangle represents time. The left vertical side of the triangle represents a part of tasks' remaining execution time, which we call the *nodal remaining execution time*, l_i, which is supposed to be consumed before each TN^k ends. Fluid schedules for each task can be constructed as overlapped in each TN^k plane, while keeping their slopes.

C. Scheduling in T-N planes

The abstraction of T-N planes is significantly meaningful in scheduling for multiprocessors, because T-N planes are repeated over time, and a good scheduling algorithm for a single T-N plane is able to schedule tasks for all repeated T-N planes. Here, good scheduling means being able to construct a schedule that allows all tasks' execution in the T-N plane to approximate the fluid schedule as much as possible. Figure 3 details the k^{th} T-N plane.

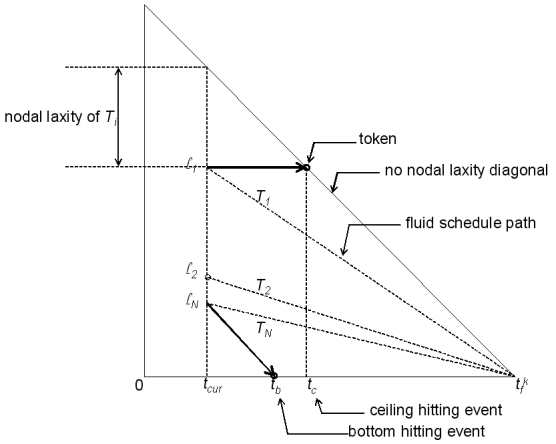

Fig. 3. k^{th} T-N Plane

The status of each task is represented as a *token* in the T-N plane. The token's location describes the current time as a value on the horizontal axis and the task's remaining execution time as a value on the vertical axis. The remaining execution time of a task here means one that must be consumed until the time t_f^k, and not the task's deadline. Hence, we call it, *nodal* remaining execution time.

As scheduling decisions are made over time, each task's token moves in the T-N plane. Although ideal paths of tokens exist as dotted lines in Figure 3, the tokens are only allowed to move in two directions. When the task is selected and executed, the token moves diagonally down, as T_N moves. Otherwise, it moves horizontally, as T_1 moves. If M processors are considered, at most M tokens can diagonally move together. The scheduling objective in the k^{th} T-N plane is to make all tokens arrive at the rightmost vertex of the T-N plane—i.e., t_f^k with zero nodal remaining execution time. We call this successful arrival, *nodally feasible*. If all tokens are made nodally feasible at each T-N plane, they are possible to be scheduled throughout every consecutive T-N planes over time, approximating all tasks' ideal paths.

For convenience, we define the *nodal laxity* of a task T_i as $t_f^k - t_{cur} - l_i$, where t_{cur} is the current time. The oblique side of the T-N plane has an important meaning: when a token

hits that side, it implies that the task does not have any nodal laxity. Thus, if it is not selected immediately, then it cannot satisfy the scheduling objective of nodal feasibility. We call the oblique side of the T-N plane *no nodal laxity diagonal* (or NLLD). All tokens are supposed to stay in between the horizontal line and the nodal laxity diagonal.

We observe that there are two time instants when the scheduling decision has to be made again in the T-N plane. One instant is when the nodal remaining execution time of a task is completely consumed, and it would be better for the system to run another task instead. When this occurs, the token hits the horizontal line, as T_N does in Figure 3. We call it the *bottom hitting event* (or event B). The other instant is when the nodal laxity of a task becomes zero so that the task must be selected immediately. When this occurs, the token hits the NLLD, as T_1 does in Figure 3. We call it the *ceiling hitting event* (or event C). To distinguish these events from traditional scheduling events such as task releases and task departures, we call events B and C *sub-events*.

To provide nodal feasibility, M of the *largest nodal remaining execution time* tasks are selected first (or LNREF) for every sub-event. We call this, the LNREF scheduling policy. Note that the task having zero nodal remaining execution time (the token lying on the bottom line in the T-N plane) is not allowed to be selected, which makes our scheduling policy non work-conserving. The tokens for these tasks are called *inactive*, and the others with more than zero nodal remaining execution time are called *active*. At time t_f^k, the time instant for the event of the next task release, the next T-N plane TN^{k+1} starts and LNREF remains valid. Thus, the LNREF scheduling policy is consistently applied for every event.

III. Synchronization for LNREF

We consider three synchronization schemes for LNREF such as wait-free buffers, lock-free objects, and lock-based objects in this section.

A. Wait-Free Buffers

Wait-free protocols generally incur additional space costs due to their multiple internal buffer usage. Fortunately, as far as the maximum number of interferences a read operation may suffer from a write operation or/and the maximum number of readers are known, the wait-free buffer algorithm guaranteeing space optimality is usable [15]. Even when either of them is not easy to obtain, the wait-free buffer algorithm is still usable by assuming the unknown value as an infinite.

We consider periodic task arrival and underload cases to compute the minimum required space cost of wait-free buffer for LNREF. It is because there exists the feasibility test of LNREF only for the periodic task sets. Though, we would like to emphasize again that the wait-free buffer algorithm is available to any case where the maximum number of interferences a read operation may suffer from a write operation or/and the maximum number of readers are known, even with non-periodic task sets.

The maximum number of interferences for each reader's operation is obtained in following Sections and the minimum space cost for wait-free buffer can be obtained by the algorithm for *Wait-Free Buffer size decision Problem* (or WFBP) presented in [15].

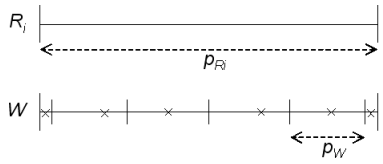

Fig. 4. Reader and Writer Execution Time Line

The maximum number of interferences that a reader may have should be computed. Let ρ denote the set of wait-free objects shared by tasks. We assume that each wait-free buffer $w_k \in \rho$ is accessed by at least two tasks and that each task accesses w_k at most once. p_i is ith task's period and deadline for periodic tasks. We assumes a writer W and multiple readers R_j access w_k. Since assuming each task accesses w_k at most once, each reader R_j and writer W for w_k corresponds to a task. Therefore, p_{R_j} and p_W imply the period of R_j and the period of W respectively. N_j^{Max} denotes the maximum number of times a writer might interfere with the jth reader during read operation.

Under underload, the worst case scenario in terms of maximum number of interferences of the reader by the writer task is illustrated in Figure 4, as presented in [13]. (We assume that each task's deadline is the same as its period.) This occurs when the first interfering write operation happens as late as possible within the writer's period and the last interfering write operation happens as early as possible within the writer's period. Each x denotes a write operation. This scenario occurs under LNREF.

Corollary 3.1 (Maximum number of interferences): When periodic tasks can be feasibly scheduled by LNREF, the upper-bound on the number of interferences N_j^{Max} that the writer W might interfere with reader R_j during read operation is:

$$N_j^{Max} = \max\left(2, 1 + \left\lceil \frac{p_{R_j}}{p_W} \right\rceil\right).$$

Proof: See [13] ■

When all N_j^{Max} for each w_k are obtained, we compute the minimum (optimal) space costs for each w_k with the algorithm of WFBP presented in [15].

However, the overload caused by relaxed task arrival is difficult to cope with. There are basic requirements for a shared object to be used for overload, i.e., the object operations should be anonymous in the sense that it does not require identities of tasks. This is because several jobs of a task may be pending in ready queue under overload and non-anonymous operations are unable to distinguish those jobs since they have the same identifier inherited from the same task. Moreover, it is difficult to assess how many jobs of a task could be pending in the ready queue at the same time.

Unfortunately, the wait-free buffer as well as other wait-free buffers are not anonymous and thus, they have those restriction for the environment allowing overload. It is intuitively understandable that the rationale behind wait-free buffer algorithm is against anonymity. Informally, wait-free buffer algorithm deploys a finite number of internal buffers proportionally to the number of readers and writers, and assigns each internal buffer to each reader and writer to avoid conflict between them, with a certain clever way to handle data replication across those internal buffers. In other words, it starts from assumption that the fixed number of readers and writers is known in advance.

B. Lock-Free Objects

Lock-free objects are a good candidate of non-blocking synchronization under overload because most well-known lock-free objects are anonymous. However, it is additionally required to assure that the lock-free retry should be bounded under overload as well as underload.

The boundness of retry is derived under several assumptions, which are made from practical observation of lock-free objects and potential interferences associated with applied scheduling algorithms as well as task arrival patterns. In [18], the observation is introduced that lock-free operations typically are very short in comparison to the length of a time quantum, that their scheduling algorithm, Pfair, is based upon. The following assumptions are the variants of theirs to bound retry for using lock-free objects with our scheduling algorithms.

(1) **Interference Assumption (IA)** Any pair of concurrent accesses to the same object may potentially interfere with each other.
(2) **Retry Assumption (RA)** A retry can be caused only by the completion of some object access. This bounds the number of retries to at most the number of concurrent accesses to an object and prevents two tasks from livelocking due to repeated mutual interferences.
(3) **Preemption Assumption (PA)** A single object access will be preempted for a finite number of times.

These assumptions help to establish bounds on retries in multiprocessor real-time systems even with interferences across processors. Holman *et al.* have bounded retry in Pfair scheduling by exploiting its tight synchrony and observation that lock-free operations take very small time [18]. However, bounding retry is more difficult for asynchronous real-time scheduling not depending on time quanta, e.g., LNREF, etc.

One example that the boundness of retry collapses is in Figure 5. It shows task T_i running on one processor in a multiprocessor system. Even if assuming (RA) holds, other tasks' scheduling events causes preemptions and hence, retry may never end. In other words, a condition for the assumption (PA) is required to hold with asynchronous real-time scheduling algorithms for use of lock-free objects.

To prevent from the continuous retry, we suggest the NPA surrounding each lock-free operation. The objective of NPA

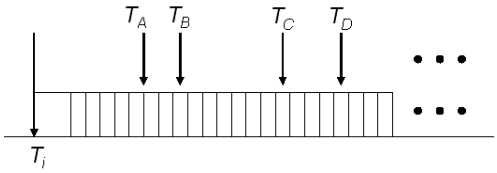

Fig. 5. An Example of Continuous Retry

is to make (PA) hold by ensuring that the access to a single object is never preempted. If there is a scheduling event during the NPA and another task tries to preempt the processor, it will be blocked up to the end of NPA.

1) Non-Preemptive Area: We follow some notations as used in [18]. Let ρ denote the set of objects shared by tasks. We assume that each object $r_k \in \rho$ is accessed by at least two tasks and that each task accesses r_k at most once. We assume that each access to r_k has a base cost of $b(r_k)$ and a retry cost of $s(r_k)$. We are assuming that the costs of all operations (e.g., read, write) of r_k are the same as in [18], which could be eliminated at the price of slightly more complicated notion and analysis. The number of tasks accessing r_k is denoted by $A(r_k)$. Note that $A(r_k)$ never exceeds the number of tasks, N, because we assume each task accesses r_k at most once. p_i is ith task's period for periodic tasks or minimum time interval between adjacent ith task's releases for sporadic tasks. e_i is the ith task's base execution cost, i.e., the worst-case cost of a single job of T_i without considering shared-object accesses. We assume a job of T_i accesses at most R_i number of shared objects. The set of shared objects that T_i accesses is denoted by γ_i. We assume M identical processors in the system.

We attempt to make three assumptions aforementioned including (IA), (RA), and (PA), with the NPA. Under all assumptions, the time cost of NPA may be extended by retry.

Theorem 3.2 (Maximum Execution Cost of NPA): When T_i accesses a lock-free object r_k with NPA-assisted sharing mechanisms, $L_{i,k}^{LF}$, the maximum execution time of the NPA is:

$$L_{i,k}^{LF} = s(r_k) \cdot \{\min(A(r_k), M) - 1\} + b(r_k),$$

where $r_k \in \gamma_i$.

Proof: NPA can be extended by continuous retry, which occur from interferences. There are $A(r_k)$ number of tasks which access r_k and thus, may cause interferences. Therefore, the maximum possible interferences for accessing r_k is $A(r_k)$ and it cannot exceed the number of processors M. Based on it, the time of possible retry is at most $s(r_k) \cdot \{\min(A(r_k), M) - 1\}$. In addition, the base access cost, $b(r_k)$, is also considered. Preemption is prohibited by the NPA surrounding each lock-free operation. ■

Note that the NPA causes blocking and thus the execution cost of the NPA affects feasibility analysis considering blocking times.

To the best of our knowledge, the feasibility analysis of more relaxed task arrival patterns than periodic one for multiprocessor real-time scheduling does not exist due to

difficulty. More specifically, most of feasibility analysis for two well-known scheduling algorithms including Pfair and global EDF have been on periodic tasks. Even though NPA-assisted lock-free approach allows sporadic tasks which may cause overloads where lock-free retry can be bounded, we also focus on the feasibility analysis of periodic tasks in following sections.

2) Lock-Free Objects For LNREF: LNREF is an optimal real-time scheduling algorithm satisfying all time requirements if the total utilization demand of tasks is less than the capacity of processors. We consider NPA-assisted lock-free sharing for LNREF.

As compared with EDF, NPA under LNREF causes more times of blocking. It is because the priority order of jobs under LNREF changes over their execution as opposed to the case where tasks are scheduable under EDF. When tasks are scheduable under EDF, the priority order of jobs never changes so that a preempted job A cannot preempt a job B that has preempted the job A before. However, it happens under LNREF that allows more scheduling events, e.g., *event C and B* as introduced in [1]. Whenever preemption occurs, blocking may arise when the preempted task is within the NPA and no processor is available. Therefore, each job can be blocked for at most $B = \max_{\forall i \forall k} \{L_{i,k}^{LF}\}$.

Theorem 3.3: (*Blocking under LNREF with NPA-assisted mechanisms*) When a set of periodic tasks T_1,...,T_N has deadline equal to period, the maximum times that T_i can be blocked within its period p_i is:

$$n_i^B = \left\lceil \frac{N+1}{2} \right\rceil \cdot \left(1 + \sum_{j=1}^{N} \left\lceil \frac{p_i}{p_j} \right\rceil \right).$$

Proof: Each T-N plane is constructed between two consecutive events of task release. The number of task releases within period p_i is $\sum_{j=1}^{N} \lceil \frac{p_i}{p_j} \rceil$. Thus, there can be at most $1 + \sum_{j=1}^{N} \lceil \frac{p_i}{p_j} \rceil$ number of T-N planes within p_i. It is known that at most $N+1$ events can occur in each T-N plane by theorems in [1]. Since blocking of T_i happens only when T_i attempts to preempt, T_i can be blocked at most $\lceil \frac{N+1}{2} \rceil$ in each T-N plane. Therefore, within the period p_i, T_i can be blocked at most $\lceil \frac{N+1}{2} \rceil \cdot (1 + \sum_{j=1}^{N} \lceil \frac{p_i}{p_j} \rceil)$ times. ■

Theorem 3.4: (*LNREF feasibility with NPA-assisted lock-free synchronization*) A set of periodic tasks T_1,...,T_N, all with deadline equal to period, is guaranteed to be feasible on M processors using LNREF if

$$\sum_{i=1}^{N} \frac{Bn_i^B + \sum_{\forall r_k \in \gamma_i} L_{i,k}^{LF} + e_i}{p_i} \leq M,$$

where $L_{i,k}^{LF} = s(r_k)(\min\{A(r_k), M\} - 1) + b(r_k)$.

Proof: When NPA-assisted lock-free objects are shared, each task's execution time extended by lock-free retry and blocking time caused by the NPA should be considered. T_i's blocking happens at most n_i^B times and each blocking time costs at most B as described in Theorem 3.2 and 3.3. Besides, when T_i accesses r_k within the NPA, it can be extended at most $L_{i,k}^{LF}$ as in Theorem 3.2. Therefore, the execution time is extended by $Bn_i^B + \sum_{\forall r_k \in \gamma_i} L_{i,k}^{LF}$ in maximum. When the total utilization demand obtained with tasks' extended execution times is less than the capacity of processors, tasks meet deadlines under LNREF. ■

Note that LNREF does not consider the case that the total utilization demand exceeds the capacity of processors. Thus, using shared objects, including lock-free shared objects, with LNREF are under the assumption that the total utilization demand never exceeds the capacity of processors.

C. Lock-Based Synchronization

We consider that shared objects implemented with queue-based spin locks within the NPA as in [6]. NPA-assisted queue lock causes blocking as aforementioned.

1) Non-Preemptive Area: As described in [6], under NPA-assisted, queue-lock based synchronization, a job may be blocked under two scenarios: (1) 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 (2) the job becomes ready when one or more lower-priority jobs are in their NPA and no processor is available.

Theorem 3.5 (Maximum Execution Cost of NPA): When T_i accesses a queue-based spin lock object r_k with NPA-assisted sharing mechanisms, $L_{i,k}^{QS}$, the maximum execution time of the NPA is:

$$L_{i,k}^{QS} = q(r_k) \cdot \min(A(r_k), M),$$

where $q(r_k)$ is the access cost to r_k, M is the number of processors in system, and $r_k \in \gamma_i$.

Proof: Since $A(r_k)$ tasks accessing r_k may cause interferences, the maximum possible interferences is $A(r_k)$ and it cannot exceed the number of processors M. Based on it, the maximum execution cost of completing access to r_k is $q(r_k) \cdot \min(A(r_k), M)$. Preemption is prohibited by the NPA surrounding each lock-free operation. ■

For the same reason we presented in Section III-B.2, we consider feasibility analysis on periodic task sets with lock-based synchronization for LNREF.

2) Lock-Based Synchronization for LNREF: We consider NPA-assisted lock-free shared objects under LNREF. More frequent scheduling events of LNREF as compared to EDF cause more blocking and thus, it affects the feasible condition. Whenever preemption occurs, blocking follows when the preempted task is within the NPA and no processor is available.

Theorem 3.6: (*LNREF feasibility with NPA-assisted lock-based synchronization*) A set of periodic tasks T_1,...,T_N, all with deadline equal to period, is guaranteed to be feasible on M processors using LNREF if

$$\sum_{i=1}^{N} \frac{Bn_i^B + \sum_{\forall r_k \in \gamma_i} L_{i,k}^{QS} + e_i}{p_i} \leq M,$$

where $L_{i,k}^{QS} = q(r_k) \cdot \min(A(r_k), M)$.

Proof: When NPA-assisted queue-lock objects are shared, each task's execution time extended by the spin queue and blocking time caused by the NPA should be considered.

T_i's blocking happens at most n_i^B times an in Theorem 3.3 and each blocking time costs at most $B = \max_{\forall i \forall k} L_{i,k}^{QS}$. Besides, when T_i accesses r_k within the NPA, it can be extended at most $L_{i,k}^{QS}$ as in Theorem 3.5. Therefore, the execution time is extended by $Bn_i^B + \sum_{\forall r_k \in \gamma_i} L_{i,k}^{QS}$ in maximum. When the total utilization demand obtained with tasks' extended execution times is less than the capacity of processors, tasks meet deadlines under LNREF. ■

D. Tradeoffs

The tradeoff between the presented wait-free method and lock-free/lock-based methods is one of space and time costs. Wait-free object sharing costs space, but incurs no additional (blocking or retry) time costs. Further, it allows the full capacity of all processors to be utilized. However, it is restricted to the case of bounded number of jobs, in contrast with lock-free and lock-based which allows unbounded number of jobs.

We now discuss the tradeoff between the lock-free and lock-based methods that we present. Lock-free object sharing does not use lock mechanisms which can potentially cause blocking, unlike lock-based object sharing. However, blocking may happen with the lock-free objects as well as the lock-based ones that we present, since both are designed to be assisted with the NPA. Despite the blocking caused by the NPA that is common to both, it is still worth discussing the tradeoffs between NPA-assisted lock-free objects and NPA-assisted queue lock objects.

We focus on the implementation of objects within the NPA (but not on the implementation of the NPA itself). First, lock-free sharing is an optimistic approach as opposed to queue-based spin locks that must lock resources even when no interference occurs. The optimism prohibits the time costs for unnecessary operations. Second, lock-free sharing mechanisms are implemented at the application level, whereas the implementation of the NPA is through kernel calls. The implementation at the application level is likely to help reduce the overhead of operations since it does not invoke kernel, which is advantageous with respect to timeliness.

For these reasons, lock-free object sharing is likely to be superior to queue-lock object sharing when simple objects (which are implementable in lock-free mechanisms) are considered. On the other hand, queue-lock object sharing is likely to be superior when more complicated objects (e.g., map, etc.) are considered.

IV. Numerical Analysis

Specifically, Theorem 3.4 and 3.6 show feasible conditions under LNREF with lock-free and lock-based synchronization, respectively. We observe that the forms of two feasibility tests are identical when the access times to shared objects are assumed similar, i.e., $s(r_k) = b(r_k) = q(r_k), \forall k$. It is because both sharing mechanisms are assisted by the NPA to ensure the boundness of access time by preventing preemption temporarily. Therefore, the performance difference between NPA-assisted lock-free objects and NPA-assisted queue-lock objects is primarily determined from the difference between their object access times. (It is not a claim generalized for every case, since NPA-assisted mechanisms considered here should not be the only approach to implement lock-free and lock-based sharing.)

In [6], they set contention-free costs for object operations in $1.3 - 6.5$ μs range for both lock-free and queue-lock approaches by their measurement. We vary the costs, $s(r_k)$, $b(r_k)$, and $q(r_k)$, from 1.0 to 6.0 to see the effect. The performance metric is the total utilization inflation denoting the difference between total utilization without shared objects and that with shared objects, which could be interpreted as synchronization overhead or utilization loss as described in [6].

We assume four processors and 20 tasks. Each task has three objects shared with others and each objects are shared by 10 tasks. Task's execution time e_i is uniformly distributed in different ranges, $[0.5, 5]$, $[1, 10]$, $[1.5, 15]$, or $[2, 20]$ ms. As Theorem 3.4 and 3.6 imply, task's execution time cost is a primary element that affects the utilization inflation. Task's period is set by e_i/u_i, where u_i is task's utilization demand. u_i is uniformly distributed in the range $(0.0, 0.2]$ and thus, the total utilization demand without shared objects never exceeds the capacity of processors. Each data was gained with 5,000 samples.

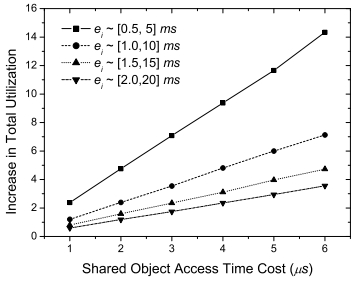

Fig. 6. LNREF-scheduled Synchronization Overhead for NPA-assisted Mechanism : Varying Tasks' Execution Time Costs

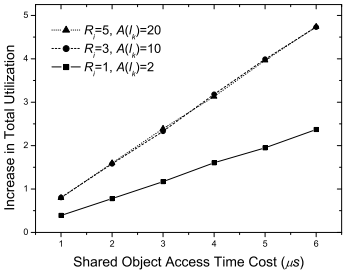

Fig. 7. LNREF-scheduled Synchronization Overhead for NPA-assisted Mechanism : Varying Number of Shared Objects

Figure 6 shows that the higher shared-object access time cost and the lower task's execution time cost cause higher increase in total utilization. It suggests that the shared-object access time cost is an important element to select between

lock-free and queue-lock shared objects under LNREF. In addition, it also implies that the selected shared object becomes more appropriate as tasks have more execution time costs since it makes the synchronization overhead negligible.

Figure 7 illustrates the increase in total utilization with varying shared-object access time cost and number of objects. As expected, the more objects are shared, the higher synchronization overhead does rise. It is also observed in Figure 7 that there is a number, more shared objects than which does not have any impact on performance as Theorem 3.4 and 3.6 indicate.

Note that the comparison results in this section show the worst case of both sharing mechanisms. Thus, the actual performance comparison by real implementation could be different.

V. Conclusions

We consider lock-based, lock-free, and wait-free synchronization methods for both LNREF. We first show the wait-free synchronization (which is appropriate for only bounded number of jobs) does not incur significant time costs, but only space costs, which helps to maintain the fairness notion of LNREF. Further, we establish the minimum (optimal) required space costs for LNREF with the space-optimal wait-free synchronization algorithm. In contrast to wait-free, lock-based and lock-free synchronization allow unbounded number of jobs and overloads. We introduce non-preemptive area to bound the time cost of lock-based and lock-free synchronization under LNREF, and derive feasible conditions for satisfying utility lower bounds. Further, we observe the tradeoff between lock-based and lock-free synchronization for LNREF, i.e., lock-free object sharing is likely to be superior to lock-based sharing when simple objects are considered, whereas lock-based object sharing is likely to be superior when more complicated objects are considered.

References

[1] H. Cho, B. Ravindran, and E. D. Jensen, "An optimal real-time scheduling for multiprocessors," in *IEEE Real-Time Systems Symposium (RTSS)*, Dec 2006.

[2] QNX, "Symmetric multiprocessing," http://www.qnx.com/products/rtos/smp.html, last accessed October 2005.

[3] S. Baruah, N. Cohen, C. G. Plaxton, and D. Varvel, "Proportionate progress: A notion of fairness in resource allocation," in *Algorithmica*, vol. 15, 1996, p. 600.

[4] U. C. Devi and J. Anderson, "Tardiness bounds for global edf scheduling on a multiprocessor," in *IEEE Real-Time Systems Symposium (RTSS)*, 2005.

[5] A. Srinivasan, P. Holman, J. Anderson, and S. Baruah, "The case for fair multiprocessor scheduling," in *IEEE International Parallel and Distributed Processing Symposium*, 2003, p. 1143.

[6] U. C. Devi, H. Leontyev, and J. Anderson, "Efficient synchronization under global edf scheduling on multiprocessors," in *Euromicro Conference on Real-Time Systems (ECRTS)*, July 2006, pp. 75–84.

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

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

[9] R. K. Clark, "Scheduling dependent real-time activities," Ph.D. dissertation, Carnegie Mellon University, 1990.

[10] H. Kopetz and J. Reisinger, "The non-blocking write protocol nbw: A solution to a real-time synchronisation problem," in *IEEE Real-Time Systems Symposium (RTSS)*, 1993, pp. 131–137.

[11] J. Chen and A. Burns, "A fully asynchronous reader/writer mechanism for multiprocessor real-time systems," CS Dept., University of York, Tech. Rep. YCS-288, May 1997.

[12] J. Anderson, R. Jain, and S. Ramamurthy, "Wait-free object-sharing schemes for real-time uniprocessors and multiprocessors," in *IEEE Real-Time Systems Symposium (RTSS)*, Dec. 1997, pp. 111 – 122.

[13] H. Huang, P. Pillai, and K. G. Shin, "Improving wait-free algorithms for interprocess communication in embedded real-time systems," in *USENIX Annual Technical Conference*, 2002, pp. 303–316.

[14] J. Anderson, S. Ramamurthy, and K. Jeffay, "Real-time computing with lock-free shared objects," *ACM Transactions on Computer Systems (TOCS)*, vol. 15, no. 2, pp. 134–165, 1997.

[15] H. Cho, B. Ravindran, and E. D. Jensen, "A space-optimal, wait-free real-time synchronization protocol," in *Euromicro Conference on Real-Time Systems (ECRTS)*, July 2005, pp. 79–88.

[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 Conf. on Measurement and Modeling of Computer Systems*, 2001, pp. 320–321.

[17] ——, "Integrating non-blocking synchronization in parallel applications: Performance advantages and methodologies," in *Proceedings of the Third Int'l Workshop on Software and Performance*, 2002, pp. 55–67.

[18] P. Holman and J. H. Anderson, "Object sharing in pfair-scheduled multiprocessor systems," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2002, pp. 111–120.

[19] V. Yadaiken, "Against priority inheritance," Finite State Machine Labs, Tech. Rep., June 2002.

[20] J. Anderson, V. Bud, and U. C. Devi, "An edf-based scheduling algorithm for multiprocessor soft real-time systems," in *Euromicro Conference on Real-Time Systems (ECRTS)*, July 2005, pp. 199–208.

[21] P. Holman and J. H. Anderson, "Adapting pfair scheduilng for symmetric multiprocessors," in *Journal of Embedded Computing*, vol. 1, no. 4, 2005, pp. 543–564.

[22] A. Chandra, M. Adler, and P. Shenoy, "Deadline fair scheduling: Bridging the theory and practice of proportionate-fair scheduling in multiprocessor servers," in *Proceedings of the 21st IEEE Real-Time Technology and Applications Symposium*, 2001, pp. 3–14.