Utility Accrual Real-Time Resource Access Protocols with Assured Individual Activity Timeliness Behavior

Abstract

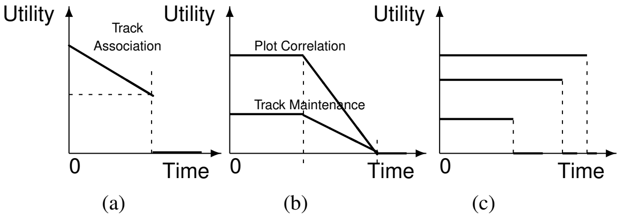
We present a class of utility accrual resource access protocols for real-time embedded systems. The protocols consider application activities that are subject to time/utility function time constraints, and mutual exclusion constraints for concurrently sharing non-CPU resources. We consider the timeliness optimality criteria of probabilistically satisfying individual activity utility lower bounds and maximizing total accrued utility. The protocols allocate CPU bandwidth to satisfy utility lower bounds; activity instances are scheduled to maximize total utility. We establish the conditions under which utility lower bounds are satisfied.

1. Introduction

Many emerging real-time embedded systems such as robotic systems in the space domain (e.g., NASA's Mars Rover [5]) and control systems in the defense domain (e.g., phased array radars [6]) operate in environments with dynamically uncertain properties. These uncertainties include transient and sustained resource overloads (due to context-dependent, activity execution times) and arbitrary, activity arrival patterns. Nevertheless, such systems desire assurances on activity timeliness behavior, whenever possible.

The most distinguishing property of such systems, is that they are subject to "soft" time constraints (besides hard). The time constraints are soft in the sense that completing an activity at any time will result in some (positive or negative) utility to the system, and that utility depends on the activity's completion time. Such soft time-constrained activities are often subject to optimality criteria such as completing all activities as close as possible to their *optimal* completion times—so as to yield maximal collective utility.

Time/utility functions [7] (TUFs) allow the semantics of soft time constraints to be precisely specified. A TUF, which generalizes the deadline constraint, specifies the utility to the system resulting from the completion of an activity as a function of its completion time. A TUF's utility values are derived from application-level QoS metrics. Figures 1(a)–1(b) show some TUF time constraints of two defense applications (see [4] and references therein for application details). Classical deadline is a binary-valued, downward "step" shaped TUF; 1(c) shows examples.



(a) (b) (c)

Figure 1: Example TUF Time Constraints. (a): AWACS *association* [4]; (b): Air Defense *correlation* & *maintenance* [4] ; (c): Step TUFs.

When activity time constraints are expressed with TUFs, the timeliness optimality criteria are often based on accrued activity utility, such as maximizing sum of the activities' attained utilities or satisfying lower bounds on activities' maximal utilities. Such criteria are called *Utility Accrual* (or UA) criteria, and scheduling algorithms that consider UA criteria are called UA scheduling algorithms.

UA criteria directly facilitate adaptive behavior during overloads, when (optimally or sub-optimally) completing more important activities, irrespective of activity urgency, is often desirable. UA algorithms that maximize summed utility under downward step TUFs (or deadlines), meet all activity deadlines during under-loads (see algorithms in [9]). When overloads occur, they favor activities that are more important (since more utility can be attained from them), irrespective of urgency. Thus, deadline scheduling's optimal timeliness behavior is a special-case of UA scheduling.

1.1. Contributions

Many embedded real-time systems involve mutually exclusive, concurrent access to shared, non-CPU resources, resulting in contention for the resources. Resolution of the contention directly affects the system's timeliness behavior.

UA algorithms that allow concurrent resource sharing exist (see [9]), but they do not provide any assurances on *individual* activity timeliness behavior—e.g., assured utility lower bounds for each activity. UA algorithms that provide assurances on individual activity timeliness behavior exist [8], but they do not allow concurrent resource sharing. No UA algorithms exist that provide individual activity timeliness assurances under concurrent resource sharing.

We solve this exact problem in this paper. We consider repeatedly occurring application activities that are subject to TUF time constraints. Activities may concurrently, but mutually exclusively, share non-CPU resources. To better account for non-determinism in task execution and inter-arrival times, we stochastically describe those properties. We consider the dual optimality criteria of: (1) probabilistically satisfying lower bounds on each activity's accrued utility, and (2) maximizing total accrued utility, while respecting all mutual exclusion resource constraints.

We present a class of lock-based resource access protocols that optimize this UA criteria. The protocols use the approach in [8] that include off-line CPU bandwidth allocation and run-time scheduling. While bandwidth allocation allocates CPU bandwidth share to tasks, scheduling orders task execution on the CPU. The protocols resolve contention among tasks (at run-time) for accessing shared resources, and bound the time needed for accessing resources.

We present three protocols, which differ in the type of resource sharing that they allow (e.g., direct, nested). We analytically establish upper bounds on the resource access times under the protocols, and establish the conditions for satisfying utility lower bounds.

Thus, the paper's contribution is the class of resource access protocols that we present. We are not aware of any other resource access protocols that solve the UA criteria that are solved by our protocols.

The rest of the paper is organized as follows: Section 2 describes our models. In Section 3, we summarize the bandwidth allocation and scheduling approach in [8] for completeness. Section 4 introduces resource sharing in this approach, and Sections 5, 6, and 7 present the protocols. In Section 8, we show a formal comparison of lock-based versus lock-free resource access protocols. We demonstrate that neither is always better than the other. We conclude in Section 9.

2. Models and Objectives

Tasks and Jobs. We consider the application to consist of a set of tasks, denoted as $\mathbf{T} = \{T_1, T_2, \cdots, T_n\}$. Each instance of a task T_i is called a job, denoted as $J_{i,j}, j \geq 1$. Jobs are assumed to be preemptible at arbitrary times.

We describe task arrivals using the Probabilistic Unimodal Arrival Model (or PUAM) [8]. A PUAM specification is a tuple $\langle p(k), w \rangle, \forall k \geq 0$, where $p(k)$ is the probability of k arrivals during any time interval w. Note that $\sum_{k=0}^{\infty} p(k) = 1$. Poisson distributions $\mathcal{P}(\lambda)$ and Binomial distributions $\mathcal{B}(n, \theta)$ are commonly used arrival distributions. Most traditional arrival models (e.g., frames, periodic, sporadic, unimodal) are PUAM's special cases [8].

We describe task execution times using non-negative random variables—e.g., gamma distributions.

A job's time constraint is specified using a TUF (jobs of a task have the same TUF). A task T_i's TUF is denoted as $U_i(t)$; thus job $J_{i,j}$'s completion at a time t will yield an utility $U_i(t)$. We focus on non-increasing TUFs, as they encompass the majority of time constraints in applications of interest to us (e.g., Figure 1).

Resource Model. Jobs can access non-CPU resources (e.g., disks, NICs, locks), which are serially reusable and are subject to mutual exclusion constraints. Similar to resource access protocols for fixed-priority algorithms [10] and for UA algorithms [9], we consider a single-unit resource model. Thus, only a single instance of a resource is present and a job explicitly specifies the desired resource. The requested time intervals for holding resources may be nested, overlapped or disjoint. Jobs are assumed to explicitly release all granted resources before the end of their execution.

Optimality Criteria. We define a *statistical* timeliness requirement for tasks. For a task T_i, this is expressed as $\langle AU_i, AP_i \rangle$, which means that T_i must accrue at least AU_i percentage of its maximum utility with the probability AP_i. This is also the requirement for each job of T_i. For e.g., if $\{AU_i, AP_i\} = \{0.7, 0.93\}$, then T_i must accrue at least 70% of its maximum utility with a probability no less than 93%. For a task T_i with a step TUF, AU_i is either 0 or 1.

We consider a two-fold optimality criteria: (1) satisfy all $\langle AU_i, AP_i \rangle$, if possible, and (2) maximize the sum of utilities accrued by all tasks.

3. Bandwidth Allocation and Scheduling

For non-increasing TUFs, satisfying a designated AU_i requires that the task's sojourn time is upper bounded by a "critical time", CT_i. Given a desired utility lower bound AU_i, $\forall t_1 \leq CT_i, U_i(t_1) \geq AU_i$ and $\forall t_2 > CT_i, U_i(t_2) < AU_i$ holds. To bound task sojourn time by CT_i, we conduct a probabilistic feasibility analysis using the processor demand approach [3]. The key to using the processor demand approach here is allocating a portion of processor bandwidth to each task. We first define *processor bandwidth*:

Definition 3.1. *If a task has a processor bandwidth* ρ*, then it receives at least* ρL *processor time during any time interval of length L.*

Once a task is allocated a processor bandwidth, the bandwidth share can be realized and enforced by a *proportional share* (or PS) algorithm (e.g., [11]). A PS algorithm can realize and enforce a desired bandwidth

ρ_i for a task T_i with a bounded allocation error, called *maximal lag*, Q, as follows: T_i will receive at least $(\rho_i L - Q)$ processor time during any time interval L. Under a PS scheme, jobs of a task execute on a "virtual CPU" that is not affected by other task behaviors. We focus on bandwidth allocation at an abstract level — using any PS algorithm with a lag Q — hereafter.

Theorem 3.1. *Suppose there are at most k arrivals of a task T during any time window of length w and all jobs of T have identical relative critical time D. Then, all job critical times can be satisfied if the underlying PS algorithm provides T with at least a processor bandwidth of* $\rho = \max\{(C+Q)/D, C/w\}$*, where C is the total execution time of k jobs released by T in a time window of w, and Q is the maximal lag of the PS algorithm.* □

Proof. Let $C_p(0,L)$ be the processor demand and $S_p(0,L)$ be the available processor time for task T_i on a time interval of $[0,L]$, respectively. The necessary and sufficient condition for satisfying job critical times is:

$$S_p(0,L) \geq C_p(0,L), \forall L > 0 \tag{3.1}$$

Let ρ be the processor bandwidth allocated to T. Thus, $S_p(0,L) = \rho L - Q$. Further, the total amount of processor time demand on $[0,L]$ is $C_p(0,L) = \left(\left\lfloor (L-D)\big/ w\right\rfloor + 1\right) C$. Therefore, Equation 3.1 can be rewritten as:

$$\rho L - Q \geq \left(\left\lfloor (L-D)\big/ w\right\rfloor + 1\right) C, \forall L > 0 \tag{3.2}$$

Since $\left(\lfloor \frac{L-D}{w} \rfloor + 1\right) \leq \left((L-D)/w + 1\right)$, it is sufficient to have $\rho L - Q \geq \left(\frac{L-D}{w} + 1\right) C, \forall L > 0$. This leads to:

$$\rho \geq \frac{C}{w} + \frac{1}{L}\left(C + Q - C\frac{D}{w}\right), \forall L > 0 \tag{3.3}$$

It is easy to see that ρ is a monotone of L. For a positive $C + Q - C\frac{D}{w}$, the maximal ρ occurs when $L = D$, which yields $\rho = (C+Q)/D$. For a negative $C + Q - C\frac{D}{w}$, the maximal ρ occurs when $L = \infty$. Combining these two cases, the theorem follows. □

For simplicity, we only consider the case $\rho \geq (C + Q)/D$, which implies $D < w$. Note that critical sections in a PS algorithm can be handled by setting Q as the longest critical section of all tasks. Let N_i be the random variable for the number of arrivals during a time window w_i. Then, the processor demand of task T_i during a time window w_i is $C_i = \sum_{j=1}^{N_i} c_{i,j}$, where $c_{i,j}$ is the execution time of job $J_{i,j}$. By Theorem 3.1, $\rho_i \geq (C_i + Q)/CT_i$, where CT_i is T_i's critical time. To satisfy the assurance probability, we require:

$$\Pr\left[\sum_{j=1}^{N_i} c_{i,j} \leq \rho_i CT_i - Q\right] \geq AP_i \tag{3.4}$$

The above condition is the fundamental bandwidth requirement for satisfying a task's critical time. If $N_i = k$, the total processor time demand during a time window becomes $\sum_{j=1}^{k} c_{i,j}$. Therefore, Equation 3.4 can be rewritten as a sum of conditional probabilities:

$$\sum_{k=0}^{\infty}\left(p_i(k) \times \Pr\left[\sum_{j=1}^{k} c_{i,j} \leq \rho_i CT_i - Q\right]\right) \geq AP_i \tag{3.5}$$

3.1. Bandwidth Solutions

Equation 3.4 can be rewritten as:

$$1 - \Pr\left[C_i \geq \rho_i CT_i - Q\right] \geq AP_i \tag{3.6}$$

By Markov's Inequality, $\Pr[X \geq t] \leq E(X)/t$ for any non-negative random variable. Therefore, $1 - \Pr[C_i \geq \rho_i CT_i - Q] \geq 1 - E(C_i)/(\rho_i CT_i - Q)$. If we can determine a ρ_i so that $1 - E(C_i)/(\rho_i CT_i - Q) \geq AP_i$, $\Pr[C_i \leq \rho_i CT_i - Q] \geq AP_i$ is also satisfied. This becomes:

$$\rho_i \geq \frac{E(C_i)}{CT_i(1 - AP_i)} + \frac{Q}{CT_i} \tag{3.7}$$

Note that N_i in Equation 3.4 is a random variable and follows a distribution specified by $p_i(a)$. By Wald's Equation, $E(C_i) = E\left(\sum_{j=1}^{N_i} c_{i,j}\right) = E(c_i)E(N_i)$. Thus,

$$\rho_i \geq \frac{E(c_i)E(N_i)}{CT_i(1 - AP_i)} + \frac{Q}{CT_i} \tag{3.8}$$

This solution is applicable for any distributions of c_i and N_i, and only requires the average number of arrivals and the average execution time.

With minimal assumption regarding task arrivals and execution times, the solution given by Equation 3.8 may be pessimistic for some distributions. Thus, an algorithm that demands and utilizes the information of *full distributions* for task arrivals and execution times is also presented in [8].

For job scheduling, [8] presents a scheduling algorithm called `UJSsched` that uses the Highest Utility Density First heuristic. `UJSsched` has the property that if all job critical times can be satisfied by EDF, then `UJSsched` is also able to do so and accrues at least the same utility as EDF does. Further, if not all job critical times can be satisfied, then `UJSsched` accrues as much utility as possible.

4. Resource Sharing With Locks

Proportional share uses large time quanta to ensure mutual exclusion. This works well for short critical sections. However, we conjecture that for some cases, a small time quantum combined with lock-based, resource access protocols may yield lower bandwidth requirement. When time quanta are smaller than the length of critical sections, preemptions of a task while it is inside a critical section may happen. Thus, we use locks to ensure mutual exclusion. With locks, three types of blocking can occur:

Direct Blocking. If a job $J_{i,m}$ requests a resource R that is currently held by another job $J_{j,k}$, we say that job $J_{i,m}$ is *directly blocked* by job $J_{j,k}$. Job $J_{j,k}$ is called the blocking job. Because processor bandwidth is allocated on a per task basis, we also say that task T_i is blocked by task T_j.

Transitive Blocking. If a job J_a is blocked by job J_b which in turn is blocked by job J_c, we say that job J_a is *transitively blocked* by J_c.

Queue Blocking. Let a set of tasks $\mathcal{TB} = \{T_{b1}, T_{b2}, \cdots, T_{bk}\}$ be simultaneously blocked on a resource R, held by task T_o. When T_o releases R, one of the blocked tasks, e.g., task T_{bm}, will acquire R and continue execution. Thus, another task T_{bn} will suffer additional blocking due to T_{bm}, besides the blocking due to T_o. We call such an additional blocking *queue blocking*, as it is caused by a queue of blocked tasks. This definition can be expanded to the case of multiple tasks in $\mathcal{TB}$ being granted R before T_{bn}.

The objective of resource access protocols is to effectively bound or reduce task blocking times. We present three protocols, called the Bandwidth Inheritance Protocol (BIP), Resource Level Policy (RLP) and the Early Blocking Protocol (EBP). BIP speeds up the execution of a blocking task and thus reduces direct blocking times. It is inspired by the Priority Inheritance Protocol (PIP) [10] in priority scheduling. RLP bounds the queue blocking time suffered by a task. However, BIP and RLP allows transitive blocking and deadlocks. EBP avoids deadlocks and bounds transitive blocking times.

Recall that `UJSsched` [8] is used to resolve competition among jobs of the same task. Thus, resource blocking can occur among jobs, which complicates the analysis of the job scheduling algorithm. Note that assurance requirements are at the task level. Thus, we simply disallow preemptions while a job holds a resource. From the perspective of the virtual processor, `UJSsched` is invoked when a new job arrives and when the currently executing job completes.

Transitive blocking and deadlocks can occur only in the presence of nested critical sections; Lemma 4.1 states this observation. Thus, BIP and RLP disallow nested sections.

Lemma 4.1. *Transitive blocking can occur only in the presence of nested critical sections. That is, if a job J_a is transitively blocked by another job J_c, there must be a job J_b that is currently inside a nested critical section.* □

Proof. By the definition of transitive blocking, there exists a job J_b that blocks J_a and is blocked by J_c. Since J_a is blocked by J_b, J_b must hold a resource, e.g., R_1. Further, the fact that J_b is blocked by J_c implies that J_b requests another resource, e.g., R_2, which is currently held by J_c. Thus, J_b must be inside a nested critical section. □

Besides the property of no transitive blocking, lack of nested critical sections also prevents deadlocks, since *hold-and-wait* — a necessary condition for deadlocks — is disallowed. We now introduce a few notations and assumptions:

- $z_{i,j}$: j^{th} critical section of task T_i;
- $d_{i,j}$: duration of critical section $z_{i,j}$ on a dedicated processor without processor contention;
- $R_{i,j}$: resource associated with critical section $z_{i,j}$;
- d_i^j: duration of task T_i's critical section that accesses resource R_i;
- $z_{i,k} \subset z_{i,m}$: $z_{i,k}$ is entirely contained in $z_{i,m}$;
- All critical sections are "properly" nested, i.e., for any pair of $z_{i,k}$ and $z_{i,m}$, either $z_{i,k} \subset z_{i,m}$, or $z_{i,m} \subset z_{i,k}$, or $z_{i,k} \bigcap z_{i,m} = \emptyset$;
- All critical sections are guarded by binary semaphores.

5. Bandwidth Inheritance Protocol

BIP's key idea is to speed up the execution time of a blocking task T, by transferring all bandwidth of tasks that are blocked by T. Thus, the blocked tasks loose their bandwidth and become stalled. We define BIP as a set of rules:

1. If a task T_i is blocked on a resource R that is currently held by a task T_j, the processor bandwidth of task T_i is inherited by task T_j. That is, the processor bandwidth of task T_j is temporarily increased to $\rho_i + \rho_j$ until T_j releases resource R. In the meanwhile, the bandwidth of task T_i becomes zero. Thus, T_i is stalled even if some jobs of T_i are eligible for execution.
2. Bandwidth inheritance is transitive. That is, if a task T_a is blocked by T_b which in turn is blocked by task T_c, then the bandwidth of T_a is also transferred to T_c.
3. Bandwidth inheritance is additive. Suppose a task T_a holds a resource R, and a set of tasks $\mathcal{TB} = \{T_i, \forall i = 1, ..., k\}$ are all blocked on R. Then, the bandwidth of T_a is increased to $\rho_a + \sum_{i=1}^{k} \rho_i$.

BIP's three rules indicate how the bandwidth of blocked tasks can be transferred to the blocking task for the three types of blocking. By doing so, we reduce the duration of the blocking task's critical section. Task bandwidth can be transferred through dynamic task *join* and *leave* operations — EEVDF allows this while maintaining a constant lag.

5.1. Blocking Time under BIP

We now upper bound a blocking task's duration of critical section. Assume that the blocking task has a total bandwidth of ρ, possibly through bandwidth inheritance. Then, the duration of the critical section is d_i/ρ. Therefore, the key to bound the duration is to lower bound the processor bandwidth allocated to a blocking task. An arbitrarily small bandwidth essentially yields an unbounded blocking time.

Section 3 presented methods to determine the minimal bandwidth needed to satisfy task utility bounds, without resource blocking. We now establish the relationship between the bandwidth requirements with and without blocking.

Theorem 5.1. *In Theorem 3.1's task model, if a task is blocked on resource access, the minimal required bandwidth is* $\rho = (B+C+Q)/D$*, where* B *is the total blocking time of jobs of the task during a time window* W. □

Proof. The proof is similar to that of Theorem 3.1 [8]. To satisfy job critical times, the available processor time during any time interval $[0,L]$, excluding the blocking time, should be greater than or equal to job processor demand:

$$S_p(0,L) - Q - \left(\left\lfloor \frac{L-D}{W} \right\rfloor + 1\right) B \geq \left(\left\lfloor \frac{L-D}{W} \right\rfloor + 1\right) C, \forall L > 0 \quad (5.1)$$

This leads to:

$$\rho L \geq \left(\left\lfloor \frac{L-D}{W} \right\rfloor + 1\right)(B+C) - Q, \forall L > 0 \quad (5.2)$$

By the same argument as in the proof of Theorem 3.1, we have $\rho \geq (B+C+Q)/D$. □

Thus, if $\rho_i^{min} = (C_i+Q)/D_i$ is T_i's processor bandwidth by assuming no resource blocking, it is safe to use ρ_i^{min} as the lower bound on T_i's bandwidth even in the presence of resource blocking. Also, observe that if T_i is a blocking task, it must inherit the bandwidth of at least one blocked task. Let $\mathcal{TR}$ be the set of tasks that may be blocked by T_i. T_i's total bandwidth while it is inside the critical section (of using resource R) is at least $\rho_i^{min} + \min\{\rho_j^{min} | j \neq i \bigwedge T_j \in \mathcal{TR}\}$. The direct blocking time caused by T_i is upper bounded by $(d_i+Q)\Big/\left(\rho_i^{min} + \min\{\rho_j^{min} | j \neq i, T_j \in \mathcal{TR}\}\right)$, where d_i is the duration of T_i's critical section for R. This blocking time calculation is repeated for all critical sections of a task, and for all jobs of a task in a time window.

5.2. Bandwidth Allocation under BIP

Let each task T_i access n_i resources, denoted $R_{i,j}, j = 1,\ldots,n_i$. Let $d_{R_{i,j}}$ denote the maximal length of the critical section for accessing resource $R_{i,j}$, and $\rho_{R_{i,j}}^{min}$ denote the smallest ρ^{min} among all tasks that may access $R_{i,j}$. T_i's direct blocking time for accessing $R_{i,j}$ is $B_{R_{i,j}} = d_{R_{i,j}}\Big/\left(\rho_{R_{i,j}}^{min} + \rho_i^{min}\right)$. A job of T_i's direct blocking time is:

$$B_D = \sum_{j=1}^{n_i} B_{R_{i,j}} = \sum_{j=1}^{n_i} \frac{d_{R_{i,j}} + Q}{\rho_{R_{i,j}}^{min} + \rho_i^{min}}, \quad (5.3)$$

where n_i is the number of critical sections of T_i. By Theorem 5.1, we require that the probability of satisfying task critical time is at least AP_i. This leads to:

$$\sum_{k=0}^{\infty} p_i(k) \Pr[B+C+Q \leq \rho_i CT_i] \geq AP_i \Rightarrow$$
$$\sum_{k=0}^{\infty} p_i(k) \Pr\left[k \sum_{j=1}^{n_i} \frac{d_{R_{i,j}}+Q}{\rho_{R_{i,j}}^{min}+\rho_i^{min}} + \sum_{j=1}^{k} c_{i,j} + Q \leq \rho_i CT_i\right] \geq AP_i \quad (5.4)$$

For all tasks, we first calculate the minimal bandwidth requirements without resource blocking, i.e., ρ_i^{min}, using the techniques in Section 3. The direct blocking time for each job of T_i, namely B_D is then calculated. Observe that the net effect of resource blocking is an increase in task execution time. In the case of direct blocking, the execution time of a job is increased by B_D, which has been calculated. Once the blocking time is calculated, the bandwidth requirement under BIP can be computed from Equation 5.4. Solutions in Section 3 can be applied to solve Equation 5.4 for ρ_i.

6. Resource Level Policy

RLP's idea is to associate a static numerical value with each task, called a task's Resource Level (or RL). A task's RL is static in the sense that it is assigned when the task is created, is maintained intact during the task's life time, and is the same for all jobs of the task. By using static RLs, we aim to produce a predictable order for accessing a shared resource, in case a queue of tasks are blocked on the same resource. Thus, queue blocking times can be bounded.

If there are n tasks in a system, the RLs of tasks are integers from 1 to n. We assume that a larger numeric value means higher RL. There are different ways for assigning static RLs. In general, static RLs must be assigned reflecting our objective of maximizing summed utility. Here, we propose several alternatives for assigning static RLs:

1. **Maximal Height of TUF.** For any pair of tasks, if $maxU_i > maxU_j$, then $RL_i > RL_j$. $maxU_i$ is the maximal height of a TUF, i.e., $maxU = \{U_i(t) | I_i \leq t \leq X_i\}$. I_i and X_i are the first and last time instances on which $U_i(t)$ is defined. The approach is easy to implement and works well for step TUFs. However, it ignores task execution time information. Further, for non-step TUFs, the maximal TUF height may be much higher than task accrued utility.
2. **Pseudo Slope.** For a task T_i, this is defined as: $pSlope_i = U_i(I_i)/(X_i - I_i)$. Pseudo Slope seeks to capture a TUF's shape, but it ignores task execution times.
3. **Pseudo Utility Density.** For a task T_i, this measures the utility that can be accrued, by average, per unit execution time: $pUD_i = U_i(\rho_i^{min} E(c_i))\Big/\rho_i^{min} E(c_i)$.

Using static RLs, the task with the highest RL will be granted a resource R if there is a queue of tasks blocked on R. Thus, when calculating the queue blocking time for task T_i, we only need to consider tasks with RLs higher than that of T_i—e.g., if $RL_i = i$, then T_i only suffers queue blocking due to tasks $T_j, j = i+1, ..., n$.

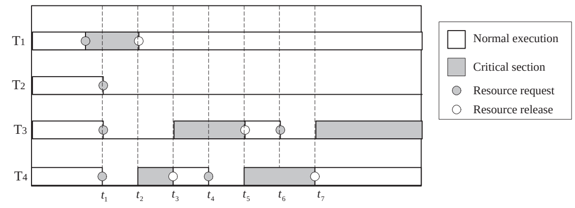


Figure 2: An Example of Using Static Resource Levels

Unfortunately, this scheme of using static RLs may yield unbounded queue blocking times for low RL tasks. Figure 2 shows an example. In Figure 2, task T_2 is blocked on a resource request and is later starved.

To overcome the difficulty with static RLs, we introduce the concept of Effective Resource Level (or ERL). Besides RL, each task is associated with an ERL, which may increase over time. The idea is to use ERL to prevent a few high RL tasks from dominating the usage of shared resources. With ERLs, RLP works as follows:

1. If a task is not blocked on any resource, its ERL is the same as its static RL.
2. Whenever a resource R is released, the ERL's of all tasks that are currently blocked on R are increased by n, where n is the number of tasks in the system.
3. When a resource R becomes free, one of the blocked tasks with the highest ERL is granted resource access. If a tie among the highest ERL tasks occurs, the task with the longest blocking time wins.
4. When a task acquires the resource on which it was blocked, its ERL returns to its static RL.

Theorem 6.1. *Under RLP, a task T_k can be queue blocked on a resource R for at most $(m-2)$ critical sections, where m is the number of tasks that may access R.* ☐

Proof. Consider a set of tasks $\mathcal{TB}$, including task T_k, that are blocked on a resource R. Obviously, $|\mathcal{TB}| \leq m-1$, because one task must be holding the resource. At time instant t_0, let R be released by the current blocking task. Thus T_k's ERL is increased to $RL_k + n$, which is higher than $RL_i, \forall i$. This high ERL effectively ensures that no tasks that are blocked on R after t_0 can queue block T_k. Therefore, T_k can only suffer additional queue blocking from existing blocked tasks, which are at most $(m-3)$ critical sections. Note that at t_0, one of the tasks from $\mathcal{TB}$ namely task T_r, is granted resource R. Therefore, the number of the remaining blocked tasks, excluding T_k, is $|\mathcal{TB} - T_k| - 1 \leq (m-3)$. The theorem follows by summing up queue blocking times before and after instant t_0, i.e., $1 + (m-3) = (m-2)$. ☐

Theorem‘6.1 leads to the following corollary:

Corollary 6.2. *The ERL of a task T_i is within the range of $[RL_i, (m-1)n + RL_i]$, where m is defined in Theorem 6.1 and n is the number of tasks in the system.* ☐

Proof. By Theorem 6.1, a task can suffer a queue blocking time of at most $(m-2)$ critical sections. In addition, it suffers one direct blocking. Upon releasing a shared resource, these blocking tasks increase the ERL of a task $(m-2)+1 = m-1$ times. Since each increase is n, the ERL of T_i is bounded by $(m-1)n + RL_i$. ☐

Theorem 6.3. *Let $\mathcal{T}_R$ be the set of tasks that may access resource R. Theorem 6.1's queue blocking time bound is tight for any $T_i \in \mathcal{T}_R$, except the highest RL task in $\mathcal{T}_R$.* ☐

Proof. Without loss of generality, let $\mathcal{T}_R = \{T_1, T_2, ..., T_m\}$ and $RL_i = i$. We prove this theorem by showing that there *always* exists a resource access pattern so that any task $T_i \in \mathcal{T}_R, i < m$ suffers a queue blocking time of $(m-2)$ critical sections. The resource access pattern can be constructed as follows: Let t_i be a time stamp and satisfies $t_{i+1} > t_i$. Now:

- t_0: Task T_{i+1} is holding resource R and tasks $\mathcal{TB} = \{T_k | T_k \in \mathcal{T}_R, k \neq i \bigwedge k \neq i+1\}$ are blocked on R. $|\mathcal{TB}| = (m-2)$.
- t_1: Task T_{i+1} releases R. A task in $\mathcal{TB}$, say T_r is granted resource R. ERL's of remaining tasks in $\mathcal{TB}$ are increased by n.
- t_2: Task T_{i+1} requests R and is blocked on R.
- t_3: Task T_i requests R and is blocked on R.

Now, at time t_3, the ERL of task T_i is lower than those of all other tasks in the blocked task queue, which includes $(m-2)$ tasks. Therefore, T_i will suffer a queue blocking time of $(m-2)$ critical sections. ☐

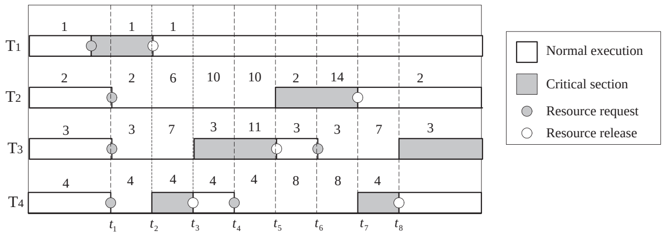


Figure 3: Dynamic Resource Levels

We now revisit the example in Figure 2. In Figure 3, we show the behavior of tasks by using the dynamic resource level adjustment rules. Note that the numbers on each timeline of a task indicates the ERL of that task. In this case, $m = 4$. Thus, task queue blocking times should be bounded by $m-2=2$ critical sections, which is consistent with Figure 3. Observe that task T_2 is queue blocked for exactly two critical sections (of T_3 and T_4, respectively). On the other hand, task T_3 suffers one critical section of queue blocking for its resource requests; task T_4 only incurs one critical section of queue blocking during its second resource request.

6.1. Queue Blocking Times under RLP

We consider a task T_b, along with a queue of k tasks, that are blocked by a task T_a. Figure 4 shows this scenario.

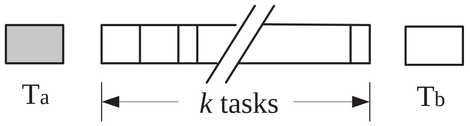


Figure 4: An Example of Queueing Blocking

To determine T_b's queue blocking time, we examine the blocking time due to each task in the $k-task$ queue. Observe that the qi^{th} task in the $k-task$ queue executes with a CPU bandwidth of at least $\rho_{qi}^{min}+\left(\sum_{j=i+1}^{k}\rho_{qj}^{min}\right)+\rho_b^{min}=\left(\sum_{j=i}^{k}\rho_{qj}^{min}\right)+\rho_b^{min}$ due to bandwidth inheritance. Thus, the total queue blocking time resulting from the k tasks is:

$$B_Q[k]=\sum_{i=1}^{k}\frac{d_{qi}+Q}{\left(\sum\limits_{j=i}^{k}\rho_{q_j}^{min}\right)+\rho_b^{min}} \tag{6.1}$$

Let $d_q=\max\{d_{qi}|i=1,...,m-2\}$ and $\rho_q^{min}=\min\{\rho_{q_j}^{min}|j=1,...,m-2\}$. Then, $B_Q[k]$ is bounded by:

$$B_Q^m[k]=\sum_{i=1}^{k}\frac{d_q+Q}{\left(\sum\limits_{j=i}^{k}\rho_q^{min}\right)+\rho_b^{min}}$$

$$=\sum_{i=1}^{k}\frac{d_q+Q}{(k-i+1)\rho_q^{min}+\rho_b^{min}}=\sum_{i=1}^{k}\frac{d_q+Q}{i\rho_q^{min}+\rho_b^{min}} \tag{6.2}$$

We need to determine a k such that $B_Q^m[k]$ achieves its maximal value and thus bounds T_b's queue blocking time. We show that the maximal queue blocking time occurs with maximal number of tasks in the queue, i.e., $k=(m-2)$.

Lemma 6.4. *The $B_Q^m[k]$ function defined in Equation 6.2 monotonically increases with k.* ☐

Proof. We define two auxiliary functions $B_Q^-[k]$ and $B_Q^+[k]$. $B_Q^-[k]$ is the amount of blocking time that may be reduced if a $(k+1)^{th}$ blocked task is added into the existing $k-task$ queue. $B_Q^+[k]$ is the additional queue blocking time due to the $(k+1)^{th}$ blocked task. That is, $B_Q^-[k]=\sum\limits_{i=1}^{k}\frac{d_q+Q}{i\rho_q^{\min}+\rho_b^{\min}}-\sum\limits_{i=1}^{k}\frac{d_q+Q}{(i+1)\rho_q^{\min}+\rho_b^{\min}}$ and $B_Q^+[k]=\frac{d_q+Q}{\rho_q^{\min}+\rho_b^{\min}}=B_Q^+$.

Now, the relationship between $B_Q^m[k+1]$ and $B_Q^m[k]$ can be derived as: $B_Q^m[k+1]=B_Q^m[k]+B_Q^+-B_Q^-[k]$. It follows that: $B_Q^-(k)/(d_q+Q)=\sum\limits_{i=1}^{k}\frac{1}{i\rho_q^{\min}+\rho_b^{\min}}-\sum\limits_{i=1}^{k}\frac{1}{(i+1)\rho_q^{\min}+\rho_b^{\min}}$

$$\begin{aligned}
&=\sum_{i=1}^{k}\left(\frac{1}{i\rho_q^{\min}+\rho_b^{\min}}-\frac{1}{(i+1)\rho_q^{\min}+\rho_b^{\min}}\right)\\
&=\frac{1}{\rho_q^{\min}+\rho_b^{\min}}-\frac{1}{2\rho_q^{\min}+\rho_b^{\min}}+\frac{1}{2\rho_q^{\min}+\rho_b^{\min}}-\frac{1}{3\rho_q^{\min}+\rho_b^{\min}}+\\
&\qquad\cdots+\frac{1}{k\rho_q^{\min}+\rho_b^{\min}}-\frac{1}{(k+1)\rho_q^{\min}+\rho_b^{\min}}\\
&=\frac{1}{\rho_q^{\min}+\rho_b^{\min}}-\frac{1}{(k+1)\rho_q^{\min}+\rho_b^{\min}}\\
&=\frac{1}{\rho_q^{\min}+\rho_b^{\min}}\frac{k\rho_q^{\min}}{(k+1)\rho_q^{\min}+\rho_b^{\min}}\\
&=\frac{1}{\rho_q^{\min}+\rho_b^{\min}}\frac{k\rho_q^{\min}}{k\rho_q^{\min}+\rho_q^{\min}+\rho_b^{\min}}<\frac{1}{\rho_q^{\min}+\rho_b^{\min}}\\
&=B_Q^+/(d_q+Q)
\end{aligned}$$

Therefore, $B_Q^m[k+1]=B_Q^m[k]+B_Q^+-B_Q^-[k]>B_Q^m[k]$. ☐

By Lemma 6.4, a task T_i's queue blocking time is $B_Q=\sum_{j=1}^{n_i}B_{Q_j}^m[m_j-2]$, where $B_{Q_j}^m[m_j-2]$ is the maximal queue blocking time for accessing resource $R_{i,j}$. Now,

$$B_{Q_j}^m[m_j-2]=\sum_{l=1}^{m_j-2}\left((d_{qj}+Q)\Big/\left(l\rho_{qj}^{min}+\rho_i^{min}\right)\right) \tag{6.3}$$

Using a technique similar to that in Equation 5.4, the bandwidth requirement under RLP is:

$$\begin{aligned}
&\sum_{k=0}^{\infty}p_i(k)\Pr[B_D+B_Q+C+Q\le\rho_iCT_i]\ge AP_i\\
&\qquad\Rightarrow\sum_{k=0}^{\infty}p_i(k)\Pr\Big[k\sum_{j=1}^{n_i}\frac{d_{R_{i,j}}+Q}{\rho_{R_{i,j}}^{min}+\rho_i^{min}}+\\
&\quad k\sum_{j=1}^{n_i}B_{Q_j}^m(m_j-2)+\sum_{j=1}^{k}c_{i,j}+Q\le\rho_iCT_i\Big]\ge AP_i
\end{aligned} \tag{6.4}$$

7. The Early Blocking Protocol

We design EBP to deal with nested critical sections. Nested sections may create deadlocks and transitive blocking. EBP's basic idea is to block an "unsafe" resource request even if the requested resource is free. An unsafe resource request is one that may cause deadlocks. Meanwhile, a safe request is granted. [2, 10] uses a similar scheme.

Let a task T invoke *nest_req_res*(R',RV) to enter a nested critical section. In their order of access, RV, called a "resource vector," is a list of resources that T may access while it is inside nested critical sections. R' is RV's first element.

For single-unit resources, a deadlock occurs if and only if there is a cycle in the resource graph. A cycle can only be formed by at least two tasks inside nested critical sections. Further, there must be at least one resource R that is requested by one task T_i and which is held by another task T_j, both of which are inside nested critical sections—i.e., the resource vectors of T_i and T_j overlap. Thus, EBP compares the resource vector of a requesting task with those of the existing tasks. If any resource vectors overlap, there is a deadlock possibility, and the requesting task is blocked.

We formulate EBP as follows: Let a task T invoke $nest\_req\_res(R', RV)$.

1. If R' is held by another task, then T is blocked.
2. If R' is free, then $nest\_req\_res(R', RV)$ may or may not be granted, per the following:
   (a) Let $\mathcal{T}_{nest}$ be the set of tasks that are currently inside nested sections. For any task $T_i \in \mathcal{T}_{nest}$, let RV_i be T_i's current resource vector.
   (b) If for any task $T_i \in \mathcal{T}_{nest}$, $RV \bigcap RV_i = \emptyset$, then $nest\_req\_res(R', RV)$ is granted; the request is blocked otherwise.
3. When a task exits a nested critical section, RLP checks if granting any pending $nest\_req\_res(R', RV)$ is safe. If more than one pending $nest\_req\_res(R', RV)$ is safe, then RLP is invoked.

7.1. Transitive Blocking Times Under EBP

We now establish that EBP is deadlock-free and can bound transitive blocking times.

Lemma 7.1. *Under EBP, for any pair of tasks that are currently inside nested critical sections, their resource vectors do not have common elements.* $\square$

Proof. Let tasks T_1 and T_2 enter nested critical sections at instants $t_1 < t_2$, respectively. If $RV_1 \bigcap RV_2 \neq \emptyset$, then T_2 cannot enter its nested section. Thus, the resource vectors of T_1 and T_2 do not have common elements. $\square$

Lemma 7.1 leads to Theorem 7.2 and Corollary 7.3:

Theorem 7.2. *EBP avoids deadlock.* $\square$

Corollary 7.3. *Under EBP, if a task T_1 is blocked by a task T_2 while T_1 is inside nested critical sections, then T_2 is not inside nested critical sections.* $\square$

Proof. Suppose T_2 is inside nested critical sections. If T_1 is blocked by T_2, then T_1 needs a resource R that is currently held by T_2. Thus, R is a common element in T_1 and T_2's resource vectors. This violates Lemma 7.1. $\square$

Theorem 7.4. *Under EBP, a chain of transitive blocking includes three tasks.* $\square$

Proof. We use $T_i \rightarrow R_i$ to denote that task T_i needs resource R_i. Similarly, $R_i \rightarrow T_i$ means that resource R_i is currently held by task T_i. Thus, a chain of transitive blocking has the form $T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_2 \rightarrow T_3 \rightarrow \ldots \rightarrow T_n$. Since there is a chain of transitive blocking, $n \geq 3$. It is easy to see that any task $T_i, i \neq 1 \bigwedge i \neq n$ must be inside nested critical sections. By Corollary 7.3, if T_2 is inside nested critical sections, T_3 cannot be inside nested critical sections. Therefore, T_3 must be at the end of the chain. Thus, $n = 3$. $\square$

Theorem 7.5. *Let a task T requests resource R_i. Let $\mathcal{T}_{i,j}$ be the set of tasks that have a resource vector $RV = \{\ldots, R_i, \ldots, R_j, \ldots\}$ and let $\mathcal{T}_j$ be the set of tasks that may access resource R_j. T's transitive blocking time for R_i is bounded by $(d_{max} + Q)\big/(\rho^{min} + \rho^{min}_{R_{i,j}} + \rho^{min}_{R_j})$. ρ^{min} is T's minimal bandwidth, $d_{max} = \max\{d^j_k | T_k \in \mathcal{T}_j\}$, $\rho^{min}_{R_{i,j}} = \min\{\rho^{min}_k | T_k \in \mathcal{T}_{i,j}\}$, and $\rho^{min}_{R_j} = \min\{\rho^{min}_k | T_k \in \mathcal{T}_j\}$.* $\square$

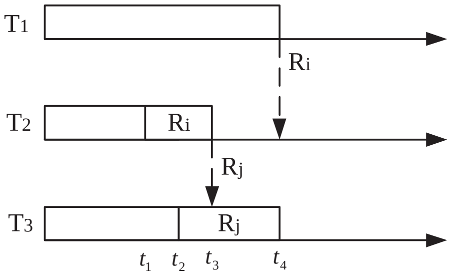


Figure 5: Illustration of Transitive Blocking

Proof. Consider a chain of transitive blocking as in Figure 5. Task T_1 is transitively blocked by task T_3 when it requests resource R_i. By Theorem 7.4, the scenario illustrated in Figure 5 is the only possible scenario.

Further, task T_3 has a bandwidth of at least $\rho^{min}_1 + \rho^{min}_2 + \rho^{min}_3$ due to bandwidth inheritance. We consider the worst case where the most pessimistic bounds are assumed. That is, $\rho^{min}_2 = \rho^{min}_{R_{i,j}} = \min\{\rho^{min}_k | T_k \in \mathcal{T}_{i,j}\}$ and $\rho^{min}_3 = \rho^{min}_{R_j} = \min\{\rho^{min}_k | T_k \in \mathcal{T}_j\}$. The theorem follows. $\square$

8. Lock-Based versus Lock-Free

As discussed earlier, our conjecture is that for some cases, our lock-based, resource access protocols may work well. For other cases, the lock-free scheme—i.e., setting quantum size as the longest critical section in the system [1], may perform better. We now explore the conditions under which resource access protocols may be beneficial, and the reverse conditions as well.

The discussion focuses on two aspects: (1) bandwidth requirement for a given task; and (2) feasibility of a task set. Given a set of n tasks and their allocated bandwidth, if $\sum_{i=1}^{n} \rho_i \leq 1$, we say that the task set is feasible for the particular bandwidth allocation. Otherwise, the task set is said infeasible for the particular allocation.

We first introduce some notations:

- ρ_i^p: bandwidth requirement of task T_i under lock-based resource access protocols;
- ρ_i^{np}: bandwidth requirement of task T_i under the lock-free scheme (also called *non-preemptive* scheme as there will be at most one preemption while a task tries to access a resource [1]);
- Q_p: quantum size under the lock-based resource access protocols
- Q_{np}: quantum size under the lock-free scheme.

Lemma 8.1. *Suppose Q_{np} equals to the length of a critical section of task T_m (accessing resource R_m). If a task T_i may be blocked on R_m, then $\rho_i^p > \rho_i^{np}$.*

Proof. Let $d_R = Q_{np}$ be the length of the critical section. If task T_i may be blocked on R, it suffers at least one direct blocking due to access to R. The direct blocking time is calculated as:

$$B_D = k\sum_{j=1}^{n_i} \frac{d_{R_{i,j}}+Q_p}{\rho_{R_{i,j}}^{min}+\rho_i^{min}} \geq \frac{d_R+Q_p}{\rho_R^{min}+\rho_i^{min}} \geq d_R+Q_p > d_R \tag{8.1}$$

The total blocking time is $B = B_D + B_Q + B_T \geq B_D > d_R$. Given the total execution time of C during a time window, we have:

$$B+C+Q_P > d_R+C+Q_p = Q_{np}+C+Q_p > Q_{np}+C \tag{8.2}$$

Recall that the fundamental bandwidth requirement under resource access protocols is:

$$\sum_{k=0}^{\infty} p_i(k)\Pr\left[B_k+C_k+Q_p \leq \rho_i^p CT_i\right] \geq AP_i \tag{8.3}$$

and under the lock-free scheme is:

$$\sum_{k=0}^{\infty} p_i(k)\Pr\left[C_k+Q_{np} \leq \rho_i^{np} CT_i\right] \geq AP_i \tag{8.4}$$

where C_k is the sum of k job execution times, B_k is the total blocking time of k jobs. Since $C_k + Q_{np} < B_k + C_k + Q_p, \forall k$, $\rho_i^{np} < \rho_i^p$. □

Lemma 8.2. *Suppose Q_{np} equals to the length of a critical section of task T_m (accessing resource R_m). If a task T_i may not be blocked on R_m, then ρ_i^p can be smaller than ρ_i^{np}.*

Proof. We prove this lemma by considering an extreme case where resource R_m is only accessed by task T_m and another task T_k. All other tasks in the system do not use any shared resources. For any task that does not use any shared resource, its blocking time is zero. Further, Q_p can be smaller than Q_{np}. Therefore,

$$B+C+Q_p = C+Q_p < C+Q_{np} \tag{8.5}$$

If that is the case, ρ_i^p is smaller than ρ_i^{np}. □

Theorem 8.3. *If a task set is feasible under the lock-free scheme, it can be infeasible under resource access protocols, and vice versa.*

Proof. We prove this theorem by examples.

1. A task set is feasible under the lock-free scheme, but infeasible using resource access protocols.

Suppose all tasks access a single resource R in a system. By Lemma 8.1, $\rho_i^{np} < \rho_i^p, \forall i = 1,...,n$. Thus,

$$\sum_{i=1}^{n} \rho_i^{np} < \sum_{i=1}^{n} \rho_i^p \tag{8.6}$$

Also assume $\sum_{i=1}^{n} \rho_i^{np} = 1$ for this particular task set. Then, $\sum_{i=1}^{n} \rho_i^p > 1$, and hence the task set if infeasible under resource access protocols.

2. A task set is feasible under resource access protocols, but infeasible under the lock-free scheme.

Consider a system where only two tasks, T_1 and T_2 need to access a resource R. Other tasks do not need to access any shared resources. Let:

$$\begin{aligned} U_p &= \sum_{i=1}^{n} \rho_i^p = (\rho_1^p + \rho_2^p) + \sum_{i=3}^{n} \rho_i^p \\ U_{np} &= \sum_{i=1}^{n} \rho_i^{np} = (\rho_1^{np} + \rho_2^{np}) + \sum_{i=3}^{n} \rho_i^{np} \end{aligned} \tag{8.7}$$

By Lemma 8.1, $\rho_i^{np} < \rho_i^p, i = 1,2$. However, if $\rho_1^p + \rho_2^p$ is small enough, we have:

$$\begin{aligned} U_p &\approx \sum_{i=3}^{n} \rho_i^p \\ U_{np} &\approx \sum_{i=3}^{n} \rho_i^{np} \end{aligned} \tag{8.8}$$

By Lemma 8.2, $\rho_i^p < \rho_i^{np}, i = 3,...,n$. Therefore, $U_p < U_{np}$. If $U_p = 1$ for this particular task set, then the task set if infeasible under the lock-free scheme. □

Through Lemmas 8.1 and 8.2 and Theorem 8.3, we demonstrate that neither the lock-free scheme, nor the resource access protocols are *always* better than the other. Specifically, if only a small number of tasks share a few resources, then using resource access protocols is beneficial. If resources are shared by most of the tasks in the system, then the lock-free scheme is more suitable in terms of bandwidth requirement.

Another hybrid case is that tasks can be partitioned into logical groups. Tasks in each logic group closely interact with each other and share resources. In addition, resource sharing across group boundaries is rare. For example, in a networked computer, device drivers may share the protocol input/output queues with the network protocol stack. On the contrary, a word processor is very unlikely to access the protocol queues. For this hybrid case, if the critical sections in a logic group are considerably longer than those in other groups, resource access protocols may still help to reduce bandwidth requirement. If all critical sections are on the same magnitude, little can be gained by using resource access protocols. Resource access protocols may even adversely affect system performance, because smaller time quanta result in higher overhead.

9. Conclusions

We present three UA resource access protocols. The protocols consider activities that are subject to TUF time constraints, and mutual exclusion constraints on sharing non-CPU resources. We consider the timeliness objective of probabilistically satisfying lower bounds on the utility accrued by each activity, and maximizing the total accrued utility. The protocols allocate CPU bandwidth to activities to satisfy utility lower bounds, while activity instances are scheduled to maximize total utility. We analytically establish the conditions under which utility bounds are satisfied.

The protocols presented here have been folded into a timing analysis software tool, in corporation with an industrial vendor. The tool is currently being used in US DoD programs. Future work includes studying the sensitivity of the protocols to the accuracy of the required scheduling parameters, and extending them to multiprocessors.

References

[1] J. H. Anderson, R. Jain, and K. Jeffay. Efficient object sharing in quantum-based real-time systems. In *IEEE RTSS*, pages 346–355, December 1998.

[2] T. P. Baker. Stack-based scheduling of real-time processes. *Real-Time Systems*, 3(1):67–99, March 1991.

[3] S. K. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems*, 2(4):301–324, Nov. 1990.

[4] R. Clark, E. D. Jensen, A. Kanevsky, J. Maurer, et al. An adaptive, distributed airborne tracking system. In *IEEE WPDRTS*, pages 353–362, April 1999.

[5] R. K. Clark, E. D. Jensen, and N. F. Rouquette. Software organization to facilitate dynamic processor scheduling. In *IEEE WPDRTS*, April 2004.

[6] GlobalSecurity.org. Multi-platform radar technology insertion program. `http://www.globalsecurity.org`.

[7] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time systems. In *IEEE RTSS*, pages 112–122, December 1985.

[8] P. Li, B. Ravindran, and E. D. Jensen. Utility accrual real-time scheduling with probabilistically assured timeliness performance. In *PARTES Workshop, ACM EMSOFT*, Sept. 2004.

[9] B. Ravindran, E. D. Jensen, and P. Li. On recent advances in time/utility function real-time scheduling and resource management. In *IEEE ISORC*, pages 55 – 60, May 2005.

[10] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Trans. Computers*, 39(9):1175–1185, Sept. 1990.

[11] I. Stoica, H. A.-Wahab, K. Jeffay, et al. A proportional share resource allocation algorithm for real-time, time-shared systems. In *IEEE RTSS*, pages 288–299, December 1996.