Utility Accrual Resource Access Protocols with Assured Timeliness Behavior for Real-Time Embedded Systems

Peng Li
Microsoft Corporation
Redmond, WA 98052, USA
pengli@microsoft.com

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

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

ABSTRACT

We present a class of utility accrual resource access protocols for real-time embedded systems. The protocols consider real-time application activities that are subject to time/utility function time constraints, and mutual exclusion constraints for concurrently accessing shared, non-CPU resources. We consider the timeliness optimality criteria of probabilistically satisfying individual activity utility lower bounds and maximizing total accrued utility. The protocols use an approach, where CPU bandwidth is allocated to activities to satisfy utility lower bounds, and activity instances are scheduled to maximize total utility. We analytically establish upper bounds on resource access blocking times, and 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 Jet Propulsion Laboratory'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. Consequently, their non-deterministic operating situations must be characterized with stochastic or extensional (rule-based) models.

The most distinguishing property of such systems, however, is that they are subject to time constraints that are "soft" (besides hard) 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. These soft time constraints are subject to optimality criteria such as completing all time-constrained activities as close as possible to their *optimal* completion times—so as to yield maximal collective utility. The optimality of the soft time constraints is generally as mission- and safety-critical as that of the hard ones.

Jensen's time/utility functions [8] (or 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 quality of service metrics. Figure 1 shows example time constraints from real applications specified using TUFs. Figures 1(a)–1(b) show some time constraints of two applications in the defense domain [3, 14]. (More real-world TUFs exist, including those with more complex shapes, but they are classified and not in the public domain.) Classical deadline is a binary-valued, downward "step" shaped TUF; 1(c) shows examples.

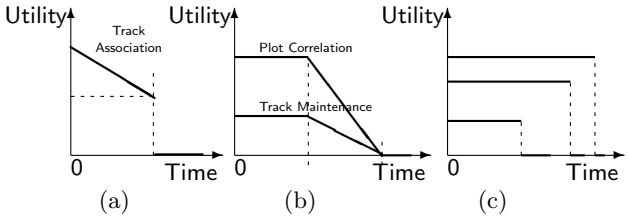

Figure 1: Example TUF Time Constraints. (a): AWACS *association* TUF; (b): Coastal Air Defense *plot correlation* & *track maintenance* TUFs; (c): Some Step TUFs.

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

Note that UA criteria directly facilitate adaptive behavior during overloads, when (optimally or sub-optimally) completing activities that are more important than those which are more urgent is often desirable. UA algorithms that maximize summed utility under downward step TUFs (or deadlines), meet all activity deadlines during under-loads [13, 4, 18]. When overloads occur, they favor activities that are more important (since more utility can be attained from them) than those which are more urgent. Thus, deadline scheduling's optimal timeliness behavior is a special-case of UA scheduling.

1.1 Contributions

Most embedded real-time systems involve mutually exclusive, concurrent access to shared, non-CPU resources (e.g., data objects), resulting in contention for the resources. Resolution of the contention directly affects the system's timeliness, and thus the system's behavior.

UA scheduling algorithms that allow concurrent sharing of resources exist. Examples include [4, 10, 18]. However, these algorithms do not provide any timeliness assurances on *individual* activity behavior. On the other hand, UA scheduling algorithms that provide timeliness assurances on individual

activity behavior exist. Examples include algorithms in [11] that provide probabilistically assured satisfaction of lower bounds on individual activities' accrued utilities. However, these algorithms do not allow concurrent resource sharing.

Thus, prior UA scheduling algorithms are concentrated on two extremes: (1) those that allow concurrent resource sharing, but provide no timeliness assurances on individual activity behavior; and (2) those that provide timeliness assurances on individual activity behavior, but without allowing concurrent resource sharing. No UA algorithms exist that bridge these extremes by providing individual activity timeliness assurances under concurrent resource sharing.

We solve this exact problem in this paper. We consider repeatedly occurring, real-time application activities (e.g., tasks) whose time constraints are specified using TUFs. 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. For such a model, we consider the dual 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 [11] 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 the resources.

We present three protocols. They differ in the type of resource sharing that they allow—e.g., direct sharing, nested sharing. We analytically establish upper bounds on the resource access times under the protocols, and thereby 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: We first introduce our models in Section 2. In Section 3, for completeness, we summarize the bandwidth allocation and scheduling approach in [11]. We introduce resource sharing in this approach, and propose protocols that bound resource access times under different sharing models in Section 4. Sections 5, 6, and 7 detail the three protocols. Finally, we conclude the paper in Section 8.

2. MODELS AND OBJECTIVES

2.1 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 can be preempted at arbitrary times.

We describe task arrivals using the Probabilistic Unimodal Arrival Model (or PUAM) in [11]. A PUAM specification is a tuple of $\langle p(k), w\rangle, \forall k \geq 0$, 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. As shown in [11], most traditional arrival models such as frame-based, periodic, sporadic, and unimodal are special cases of PUAM.

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. The TUF of a task T_i is denoted as $u_i(t)$; thus completion of a job $J_{i,j}$ of task T_i at a time t will yield an utility $u_i(t)$.

We focus on non-increasing TUFs—i.e., those TUFs for which utility never increases as time advances, as they encompass the majority of time constraints in applications of interest to us (e.g., Figure 1).

2.2 Resource Model

Jobs can access non-CPU resources, which in general, are serially reusable. Examples include physical resources (e.g., disks) and logical resources (e.g., critical sections guarded by mutexes).

Similar to fixed-priority resource access protocols (e.g., priority inheritance, priority ceiling) [15] and that for UA algorithms [4, 10], we consider a single-unit resource model. Thus, only a single instance of a resource is present and a job must explicitly specify the resource that it wants to access.

Resources can be shared and can be subject to mutual exclusion constraints. A job may request multiple shared resources during its lifetime. The requested time intervals for holding resources may be nested, overlapped or disjoint. We assume that a job explicitly releases all granted resources before the end of its execution.

Jobs of different tasks can have precedence constraints. For example, a job J_k can become eligible for execution only after a job J_l has completed, because J_k may require J_l's results. As in [4, 10], we model such precedences as mutually exclusive resource sharing.

2.3 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 possible utility with the probability AP_i. This is also the requirement for each job of T_i. For example, 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 can only take the value 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. Note that this only expresses the minimal requirement. A stronger timeliness assurance, such as higher accrued utility or higher assurance probability is generally desired.

3. CPU BANDWIDTH ALLOCATION AND SCHEDULING APPROACH

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. Once the requirement of accruing AU_i is converted to bounding task sojourn time by CT_i, a probabilistic feasibility analysis similar to that for deadlines can be conducted.

We consider the processor demand analysis approach [2]. The key to using 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 [16, 9, 12]. 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. Furthermore, under a PS scheme, jobs of a task execute on a "virtual processor" that is not affected by the behavior of other tasks. Thus, highly dynamic and efficient UA scheduling can be performed on jobs. Hence, we focus on bandwidth allocation at an abstract level — using any PS algorithm with a maximal 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 $C_p(0, L) = \left\lfloor (L - D)\Big/ w \right\rfloor + 1\ \ C$. Therefore, Equation 3.1 can be rewritten as:

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

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

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

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 $D < w$. Furthermore, 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 of w_i. Then, the processor demand of task T_i during a time window of 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 the critical time of task T_i. 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:

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

3.1 A General Solution

The feasibility condition (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, $E(C_i) = E\ \ \sum_{j=1}^{N_i} c_{i,j}\ \ = E(c_i)E(N_i)$. 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.

3.2 A Binary Search Strategy

With minimal assumption regarding task arrivals and execution times, the solution given by Equation 3.8 may be pessimistic for some distributions. We now present a binary search strategy that demands and utilizes the information of full distributions for task arrivals and execution times.

For brevity, we introduce a shorthand notation for the left hand side of Equation 3.5, as follows: Let $feasibleProb_i(\rho_i) = \sum_{k=0}^{\infty}\ p_i(k) \times \Pr\left[\sum_{j=1}^{k} c_{i,j} \leq \rho_i CT_i - Q\right]$. This function calculates the probability of satisfying all job critical times if task T_i were assigned a processor bandwidth of ρ_i.

Now, Equation 3.5 can be restated as solving the minimal ρ_i that satisfies $feasibleProb_i(\rho_i) \geq AP_i$. Note that function $feasibleProb_i(\rho_i)$ is a monotone in terms of ρ_i. Therefore, a binary search is applicable on the set of $\rho_i \in [0, 1]$.

The binary search strategy, presented in Algorithm 1 works as follows: The algorithm accepts an error bound ϵ, a range to search the minimal bandwidth, denoted as $[a, b]$, and the required assurance probability AP_i. In our case, invoking $minBW(\epsilon, 0, 1, AP_i)$ returns either the minimal required ρ_i, or **failure** if even the maximal processor bandwidth i.e., $\rho_i = 1$, cannot satisfy AP_i. In the worst case, function $minBW(\epsilon, a, b, AP_i)$ performs $\log_2((b - a)/\epsilon)$ searches. If $a = 0$ and $b = 1$, then the worst case complexity of $minBW(\epsilon, a, b, AP_i)$ becomes $\log_2(1/\epsilon)$.

Let $S_k = \sum_{j=1}^{k} c_{i,j}$. Given a task arrival pattern $\langle p_i(a), w_i \rangle$, the key to using Algorithm 1 is to calculate the sum distribution $\Pr[S_k \leq \rho_i CT_i]$. Computing the sum distribution of

Input : ϵ, a, b, AP_i
Output : the minimal ρ_i that satisfies $feasibleProb(\rho_i) \geq AP_i$ for task T_i; **failure** if even the maximal bandwidth b cannot satisfy AP_i
if $feasibleProb_i(b) < AP_i$ **then**
  return failure;
if $b - a \leq \epsilon$ **then**
  return a;
if $feasibleProb_i((a+b)/2) \geq AP_i$ **then**
  return `minBW` $(\epsilon, a, (a+b)/2, AP_i)$;
else
  return `minBW` $(\epsilon, (a+b)/2, b, AP_i)$;

Algorithm 1: `minBW`(ϵ, a, b, AP_i) Function

a set of independent random variables, in general, requires convolutions. In practice, convolutions are performed for small k. When k is large, the sum distribution can be approximated by the Central Limit Theorem (CLT), regardless of the original distribution of job execution times. The Central Limit Theorem states that when k is large, S_k converges to a normal distribution. Further, if $E\ |c_i - E(c_i)|^3\ < \infty$, the error of using CLT to approximate the sum distribution is bounded by the Berry-Esséen Theorem.

3.3 An UA Scheduling Algorithm

The EDF optimality is only meaningful for satisfying job critical times — it does not maximize total utilities, which is one of our objectives. Thus, we develop a UA job scheduling algorithm, called `UJSsched`, with the following properties:

- If all job critical times can be satisfied by EDF, then `UJSsched` should be able to do so and accrue at least the same utility as EDF does; and
- If not all job critical times can be satisfied, `UJSsched` should seek to accrue as much utility as possible.

We desire a fast job scheduling algorithm. This is particularly true in the context of PS, where the PS mechanism itself may be implemented by another scheduling algorithm, such as EEVDF [16]. Thus, we adopt the Highest Utility Density First (HUDF) heuristic as a way to improve accrued utility. Our rational for doing so is because HUDF is easy to implement, incurs small overhead, and has been shown to yield high utility during both underloads and overloads [10].

Let t_0 be the time instant when a scheduling event occurs. The Utility Density (UD) of a job J is defined as the ratio of its utility at its predicted completion time to its remaining execution time—i.e., $UD = u\left(t_0 + c(t_0)\right)/c(t_0)$, where $u()$ is J's TUF and $c(t_0)$ is J's remaining execution time at t_0.

`UJSsched`'s design closely follows its desired properties. As shown in Algorithm 2, the algorithm first examines if the set of ready jobs are schedulable under both EDF and HUDF, assuming jobs are executed on a processor with an execution rate of ρ_i. If they are, the highest utility density job is selected. If the jobs are only schedulable under EDF, then the earliest-critical-time job is selected. If the jobs are not schedulable, then the next job is selected by HUDF.

Given n jobs, `UJSsched`'s worst-case complexity is $O(n)$. `UJSsched` has also the following important property:

LEMMA 3.2. *If all job critical times can be satisfied by EDF, then* `UJSsched` *is able to do so and accrues at least the same utility as EDF does.*

This lemma directly follows Algorithm 2.

Input : a queue of ready jobs, denoted as RQ
Output : job J_s to be executed next; or $NULL$
if *RQ is empty* **then**
  Select $NULL$ job;
Let U_d be the accrued utilities of all jobs in RQ under EDF;
Let U_h be the accrued utilities of all jobs in RQ under Highest Utility Density First (HUDF);
if *RQ is feasible under EDF* **then**
  if *RQ is feasible under HUDF and* $U_h > U_d$ **then**
    Select the highest utility density job;
  else
    Select the earliest critical time job;
else
  Select the highest utility density job;

Algorithm 2: `UJSsched` Algorithm

4. BANDWIDTH ALLOCATION UNDER SHARED RESOURCES

Proportional share uses large time quanta to ensure mutual exclusion. Using large time quanta 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 a lock-based approach to ensure mutual exclusion—i.e., a job is required to acquire the lock guarding a shared resource R before it can use R.

Due to the use of locks, three types of blocking may occur:

1. **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.
2. **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.
3. **Queue Blocking.** Suppose a set of tasks $\mathcal{TB} = \{T_{b1}, T_{b2}, \cdots, T_{bk}\}$ are 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 time due to T_{bm}, besides the blocking time 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 the blocking time suffered by a task. Toward this, we present three protocols. The first protocol is called Bandwidth Inheritance (or BWI). The BWI protocol speeds up the execution of a blocking task and thus reduces direct blocking times. The second protocol, Resource Level Policy (or RLP), can bound the queue blocking time suffered by a task. However, neither BWI nor RLP can solve the problem of transitive blocking. Furthermore, without additional mechanisms, deadlocks may occur. Therefore, a third protocol, called Early Blocking Protocol (or EBP) is proposed. The EBP protocol can avoid deadlocks as well as bound the duration of transitive blocking.

Recall that we use the `UJSsched` algorithm (Section 3.3) to resolve competition among jobs of the same task. Therefore, resource blocking can occur among jobs, which complicates the analysis of the job scheduling algorithm. Furthermore, note that assurance requirements are at the task level. Therefore, we simply disallow preemptions while a job holds a resource. From the perspective of the virtual processor, the `UJSsched` algorithm is invoked when a new job arrives and when the currently executing job completes.

Note that both transitive blocking and deadlock can occur only in the presence of nested critical sections. We state this observation in Lemma 4.1. Thus, both BWI and RLP protocols only consider the case of no nested critical 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. Furthermore, 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, because one of the necessary condition for deadlocks, namely *hold-and-wait* is not possible. Therefore, the BWI and RLP protocols are suitable for the cases of no nested critical sections.

We now introduce a few notations and assumptions:

- $z_{i,j}$ denotes the j^{th} critical section of task T_i;
- $d_{i,j}$ denotes the duration of critical section $z_{i,j}$ on a dedicated processor without processor contention;
- $R_{i,j}$ is the resource associated with critical section $z_{i,j}$;
- d_i^j denotes the duration of task T_i's critical section that accesses resource R_i;
- $z_{i,k} \subset z_{i,m}$ means that $z_{i,k}$ 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. Thus, only one job can be inside a critical section at any given time.

5. BANDWIDTH INHERITANCE PROTOCOL

The basic idea of the BWI protocol is to speed up the execution time of a blocking task, e.g., task T, by transferring all bandwidth of tasks that are blocked by T. Consequently, the blocked tasks loose their bandwidth and thus are stalled.

5.1 Protocol Definition

We define the BWI protocol 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 $\rho_a + \sum_{i=1}^{k} \rho_i$.

Recall that we have identified three types of blocking: direct blocking, transitive blocking, and queue blocking. The three rules of the BWI protocol explicitly indicate how the bandwidth of blocked tasks can be transferred to the blocking task for the three types of blocking. By doing so, we can reduce the duration of the blocking task's critical section.

Transferring of task bandwidth can be implemented through dynamic task *join* and *leave* operations. The ability to allow tasks to join and leave a system while maintaining a constant lag is one of the unique features of EEVDF.

5.2 Blocking Time under BWI

Having defined the BWI protocol, one problem remains unsolved: What is the upper bound on the blocking task's duration of critical section? Given a duration of d_i on a dedicated processor, the answer obviously depends on how much bandwidth the blocking task has.

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.

In Section 3, we developed methods to calculate the minimal bandwidth to satisfy a task's timeliness (utility) requirements, assuming no resource blocking. In the following theorem, we establish the relationship between the bandwidth requirements with and without resource blocking.

THEOREM 5.1. *For the same task model assumed in Theorem 3.1, if a task is blocked on resource access, then the minimal required bandwidth is $\rho = (B + C + Q) \;\; D$, where B is the total blocking time suffered by jobs of the task during a specification time window of W.*

PROOF. The proof is similar to that of Theorem 3.1. 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:

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

This leads to:

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

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

Therefore, 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. Further, 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 $(d_i + Q) \Big/ \;\; \rho_i^{min} + \min\{\rho_j^{min} | j \neq i, T_j \in \mathcal{TR}\}$,

where d_i is the duration of T_i's critical section of using resource R. This blocking time calculation is repeated for all critical sections of a task, and for all jobs of a task in a specification time window.

5.3 Bandwidth Allocation under BWI

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 $\rho^{min}_{R_{i,j}}$ denote the smallest ρ^{min} among all tasks that may access $R_{i,j}$. The direct blocking time that T_i may suffer for accessing $R_{i,j}$ is:

$$B_{R_{i,j}} = \frac{d_{R_{i,j}}}{\rho^{min}_{R_{i,j}} + \rho^{min}_i} \tag{5.3}$$

The direct blocking time that a job of T_i may suffer is:

$$B_D = \sum_{j=1}^{n_i} B_{R_{i,j}} = \sum_{j=1}^{n_i} \frac{d_{R_{i,j}} + Q}{\rho^{min}_{R_{i,j}} + \rho^{min}_i}, \tag{5.4}$$

where n_i is the number of critical sections that T_i may use.

According to Theorem 5.1, we require that the probability of satisfying task critical time is at least AP_i. This leads to:

$$\begin{aligned} &\sum_{k=0}^{\infty} p_i(k) \Pr[B + C + Q \leq \rho_i CT_i] \geq AP_i \\ &\Rightarrow \sum_{k=0}^{\infty} p_i(k) \Pr\left[k \sum_{j=1}^{n_i} \frac{d_{R_{i,j}}+Q}{\rho^{min}_{R_{i,j}}+\rho^{min}_i} + \sum_{j=1}^{k} c_{i,j} + Q \leq \rho_i CT_i \right] \\ &\geq AP_i \end{aligned} \tag{5.5}$$

For all tasks, we first calculate the minimal bandwidth requirements without resource blocking, i.e., ρ^{min}_i, 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 BWI can be computed from Equation 5.5. Solutions in Section 3 can be applied to solve Equation 5.5 for ρ_i.

6. RESOURCE LEVEL POLICY

6.1 Protocol Definition

We develop RLP to specifically bound queue blocking times. The idea is to associate a static numerical value with each task, called Resource Level (or RL) of the task. 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. Therefore, 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. However, there are different ways of assigning static RLs. In general, the way of assigning static RLs should reflect our goal of utility accrual, i.e., maximizing the accrued utilities. Therefore, we propose several criteria for assigning static RLs:

(1) **Maximal Heights of TUFs.** 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., $maxU = \{U_i(t) | I_i < t < X_i\}$. The approach is easy to implement and works well for step TUFs. However, it ignores the task execution time information. Furthermore, for non-step TUFs, the maximal height of a TUF may be much higher than that can be accrued by a task.

(2) **Pseudo Slopes.** The pseudo slope of a task is defined as: $pSlope_i = U_i(I_i)/(X_i - I_i)$. The pseudo slope metric seeks to capture the shape information of a TUF. Still, task execution time is ignored from the calculation.

(3) **Pseudo Utility Densities.** The pseudo utility density of a task measures the amount of utility that the task can accrue, by average, per unit time of execution: $pUD_i = \frac{U_i(\rho^{min}_i E(c_i))}{\rho^{min}_i E(c_i)}$.

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. Therefore, when calculating the queue blocking time for task T_i, we only need to consider tasks with RLs higher than that of T_i. For example, if $RL_i = i$, then task T_i only suffers queue blocking due to tasks $T_j, j = i+1, ..., n$.

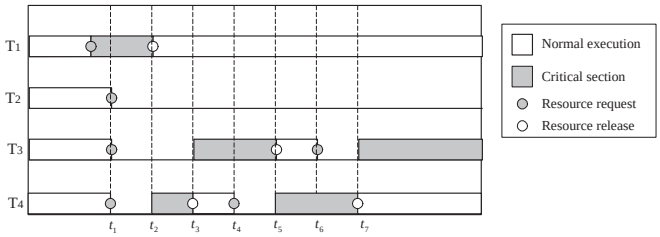

Figure 2: An Example of Using Static Resource Levels

Unfortunately, this scheme of using static RLs may yield unbounded queue blocking times for low RL tasks. Figure 2 shows an example of why static resource levels are not adequate. 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 resource level policy, a task T_k can be queue blocked on a resource R for at most the duration of $(m-2)$ 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. Assume that at time instant t_0, resource R is 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 $\mathcal{TB}$, i.e., T_m, 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., $1 + (m-3) = (m-2)$.

The above proof also holds in the presence of multiple resources. Observe that a task T_m that is blocked on a resource R after t_0 must not be blocked on any other resources when it requests R. Therefore, by the RLP protocol, its ERL must be the same as its RL, which is lower than that of T_k. This argument is true even if T_m holds other resources. □

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. According to 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. The bound on queue blocking time in Theorem 6.1 is tight for any task $T_i \in \mathcal{T}_R$, 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. □

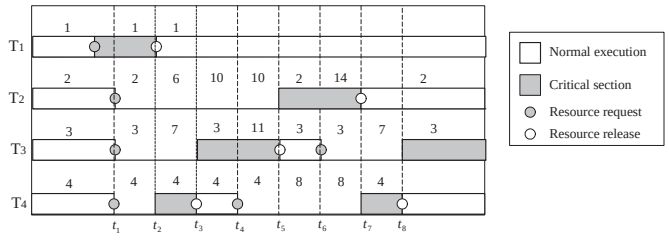

Figure 3: Dynamic Resource Levels

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.

6.2 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. This scenario is illustrated in Figure 4.

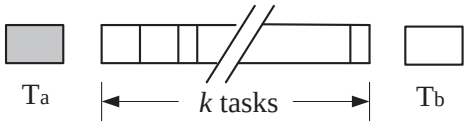

Figure 4: An Example of Queueing Blocking

To calculate the queue blocking time for T_b, 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 processor bandwidth of at least $\rho_{qi}^{min} + \sum_{j=i+1}^{k} \rho_{qj}^{min} + \rho_b^{min} = \sum_{j=i}^{k} \rho_{qj}^{min} + \rho_b^{min}$ due to bandwidth inheritance. Thus, the total queue blocking time resulting from the k tasks is:

$$B_Q(k) = \sum_{i=1}^{k} \frac{d_{qi} + Q}{\sum_{j=i}^{k} \rho_{qi}^{min} + \rho_b^{min}} \tag{6.1}$$

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

$$B_Q^m(k) = \sum_{i=1}^{k} \frac{d_q + Q}{\sum_{j=i}^{k} \rho_q^{min} + \rho_b^{min}}$$

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

Now, we need to determine a k such that $B_Q^m(k)$ achieves its maximal value and therefore bounds task T_b's queue blocking time. We show that the maximal queue blocking time occurs when there are the 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, $B_Q^-(k) = \sum_{i=1}^{k} \frac{d_q+Q}{i\rho_q^{\min}+\rho_b^{\min}} - \sum_{i=1}^{k} \frac{d_q+Q}{(i+1)\rho_q^{\min}+\rho_b^{\min}}$ and $B_Q^+(k) = \frac{d_q+Q}{\rho_q^{\min}+\rho_b^{\min}} = B_Q^+$.

By using $B_Q^-(k)$ and B_Q^+, one can derive the relationship between $B_Q^m(k+1)$ and $B_Q^m(k)$:

$$B_Q^m(k+1) = B_Q^m(k) + B_Q^+ - B_Q^-(k) \tag{6.3}$$

Note that:

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

Therefore, $B_Q^m(k+1) = B_Q^m(k) + B_Q^+ - B_Q^-(k) > B_Q^m(k)$. □

By Lemma 6.4, the queue blocking time of a task T_i is:

$$B_Q = \sum_{j=1}^{n_i} B_{Q_j}^m(m_j - 2) \tag{6.4}$$

where $B_{Q_j}^m(m_j-2)$ is the maximal queue blocking time for accessing resource $R_{i,j}$. Now,

$$B_{Q_j}^m(m_j-2) = \sum_{l=1}^{m_j-2} (d_{qj}+Q) \Big/ \ \ l\rho_{qj}^{min} + \rho_i^{min} \tag{6.5}$$

Using a technique similar to that in Equation 5.5, the bandwidth requirement under RLP is:

$$
\begin{aligned}
&\sum_{k=0}^{\infty} p_i(k) \Pr[B_D + B_Q + C + Q \le \rho_i C T_i] \ge AP_i \\
&\qquad \Rightarrow \sum_{k=0}^{\infty} p_i(k) \Pr\Big[k \sum_{j=1}^{n_i} \frac{d_{R_{i,j}}+Q}{\rho_{R_{i,j}}^{min}+\rho_i^{min}} + \\
&k \sum_{j=1}^{n_i} B_{Q_j}^m(m_j-2) + \sum_{j=1}^{k} c_{i,j} + Q \le \rho_i C T_i\Big] \ge AP_i
\end{aligned} \tag{6.6}
$$

7. THE EARLY BLOCKING PROTOCOL

We design EBP to specifically deal with nested critical sections. Nested critical sections create two problems: deadlocks and transitive blocking.

One common approach for handling nested critical sections is to disallow them. If critical sections $z_{i,j} \subseteq z_{i,k}$, then disallowing nested critical sections requires that resources $R_{i,j}$ and $R_{i,k}$ are requested and held simultaneously over the complete duration of $z_{i,k}$. The advantages are easy implementation and deadlock freedom. The disadvantage is reduced concurrency. If $z_{i,k}$ is considerably longer than $z_{i,j}$, then concurrency will be significantly reduced. Thus, in general, disallowing nested critical sections is reasonable if critical sections are short and preemptions are rare. If that is the case, little can be gained by allowing nested critical sections. The EDF/DDM algorithm [7] is based on this premise.

Our research is motivated by the fact that not all critical sections are short. For example, the time to access some physical devices in applications of interest to us (e.g., Rover robot) can be in the magnitude of milliseconds or longer. Thus, we develop EBP. We show that EBP avoids deadlocks as well as bound the duration of transitive blocking.

7.1 Protocol Definition

The 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. Hence, the name "early" blocking reflects this approach. Meanwhile, a safe request is granted. A similar scheme is used in [15, 1].

Assume that a task T invokes $nest\_req\_resource(R', RV)$ for requesting to enter nested critical sections. In their order of access, RV is a list of resources that T may access while it is inside nested critical sections. We call RV a "resource vector." R' is the first element in RV.

For single-unit resources, as we assume in this work, the necessary and sufficient condition for a deadlock is the existence of a cycle in the resource graph. A cycle can only be formed by at least two tasks inside nested critical sections. Furthermore, 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. In other words, the resource vectors of T_i and T_j overlap. Therefore, EBP compares the resource vector of a requesting task with those of the existing tasks. If any overlap of resource vectors exists, there is a possibility of deadlock. Thus, the requesting task is blocked.

Let task T invoke the request $nested\_req\_resource(R', RV)$. We formulate EBP as follows:

1. If resource R' is currently held by another task, then task T is blocked.
2. If resource R' is free, then $nest\_req\_resource(R', RV)$ may or may not be granted, per the following:
   (a) Let $\mathcal{T}_{nest}$ be the set of tasks that are currently inside nested critical 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 $nest\_req\_resource(R', RV)$ is granted; the request is blocked otherwise.
3. Whenever a task exits a nested critical section, the protocol is invoked to check if granting any pending $nest\_req\_resource(R', RV)$ is safe. If more than one pending $nest\_req\_resource(R', RV)$ is safe, then RLP is invoked.

7.2 Properties and Transitive Blocking Times

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.*

PROOF. Suppose two 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 critical section. Thus, the resource vectors of T_1 and T_2 do not have common elements.

It seems that a race condition may arise if $t_1 = t_2$. However, since we only consider uni-processor systems, T_1 and T_2 cannot make resource requests at the same time, though simultaneous resource requests may occur on virtual CPUs. □

THEOREM 7.2. *EBP avoids deadlock.*

PROOF. We prove this theorem by contradiction. Suppose deadlock occurs under EBP. Then, a cycle exists in the resource graph, as shown in Figure 5.

Note that *all* tasks on the cycle are inside nested critical sections. Furthermore, any pair of adjacent tasks on the cy-

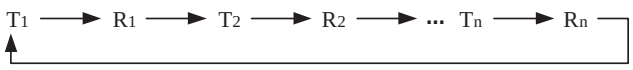

Figure 5: A Cycle in a Resource Graph

cle have a common element in their resource vectors, which contradicts Lemma 7.1. The theorem follows. □

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.*

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 phenomena violates Lemma 7.1. □

THEOREM 7.4. *Under EBP, a chain of transitive blocking includes three tasks.*

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 $T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_2 \rightarrow T_3 \rightarrow \ldots \rightarrow T_n$. 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, $n = 3$. □

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

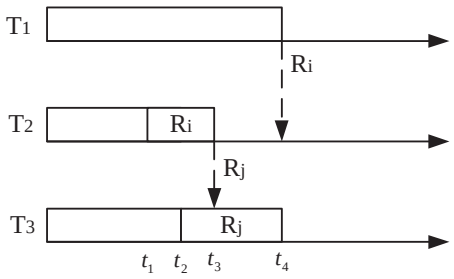

Figure 6: Illustration of Transitive Blocking

PROOF. Consider a chain of transitive blocking as in Figure 6. 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 6 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 $\rho^{min}_3 = \rho^{min}_{R_j} = \min\{\rho^{min}_k | T_k \in \mathcal{T}_j\}$. These bounds lead to the theorem. □

8. CONCLUSIONS

In this paper, we present three resource access protocols for real-time embedded systems. The protocols consider real-time activities that are subject to TUF time constraints, and share non-CPU resources under mutual exclusion constraints. We consider the timeliness objective of probabilistically satisfying lower bounds on the utility accrued by each activity, besides maximizing the total accrued utility. The protocols follow an approach where CPU bandwidth is allocated to activities to satisfy utility lower bounds, and activity instances are scheduled to maximize total utility. The protocols bound the time incurred by activities for accessing shared resources. We analytically establish upper bounds for resource access times, and consequently establish the conditions under which utility bounds are satisfied.

The protocols presented here have been incorporated into a timing analysis software tool, called *Virginia Tech TUF/UA Design Toolkit*, in corporation with Tri-Pacific Software, Inc. The toolkit has been transitioned to, and is currently being used in, programs MITRE is engaged in for the US DoD and other U.S. Government agencies. Ongoing work is developing a commercial version of this toolkit by integrating it with Tri-Pacific's RapidRMA [17].

Acknowledgements

This work was sponsored by the US Office of Naval Research under Grant N00014-00-1-0549 and The MITRE Corporation under Grant 52917.

9. REFERENCES

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

[2] 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, November 1990.

[3] 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.

[4] R. K. Clark. *Scheduling Dependent Real-Time Activities*. PhD thesis, Carnegie Mellon University, 1990. CMU-CS-90-155.

[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] K. Jeffay. Scheduling sporadic tasks with shared resources in hard-real-time systems. In *IEEE RTSS*, pages 89–99, December 1992.

[8] 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.

[9] M. B. Jones, D. Roşu, and M.-C. Roşu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *ACM SOSP*, pages 198–211, 1997.

[10] P. Li. *Utility Accrual Real-Time Scheduling: Models and Algorithms*. PhD thesis, Virginia Tech, 2004. `http://scholar.lib.vt.edu/theses/available/etd-08092004-230138/`.

[11] P. Li, H. Wu, B. Ravindran, and E. D. Jensen. On utility accrual resource management with assured timeliness behavior. `http://www.ee.vt.edu/~realtime/ECRTS05.pdf`, December 2004. Under review at ECRTS 2005.

[12] G. Lipari, J. Carpenter, and S. Baruah. A framework for achieving inter-application isolation in multiprogrammed, hard real-time environments. In *IEEE RTSS*, pages 217–226, 2000.

[13] C. D. Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, Carnegie Mellon University, 1986. CMU-CS-86-134.

[14] D. P. Maynard, S. E. Shipman, et al. An example real-time command, control, and battle management application for alpha. Technical Report Archons Project Technical Report 88121, Dept. of Computer Science, Carnegie Mellon University, December 1988.

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

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

[17] Tri-Pacific Software, Inc. Rapid rma. `http://www.tripac.com/NEW/prod-fact-rrm.html`. Last updated, October 1999.

[18] H. Wu, B. Ravindran, E. D. Jensen, and U. Balli. Utility Accrual Scheduling under Arbitrary Time/utility Functions and Multi-unit Resource Constraints. In *RTCSA*, pages 80–98, August 2004.