

# Response Time Analysis for Distributed Real-Time Systems with Bursty Job Arrivals

Chengzhi Li

Riccardo Bettati

Wei Zhao

Department of Computer Science

Texas A & M University

College Station, TX 77843-3112

Phone 409 - 845 - 5098

Email: {chengzhi,bettati,zhao}@cs.tamu.edu

## Abstract

*This paper presents a new schedulability analysis methodology for distributed hard real-time systems with bursty job arrivals. The schedulability is analyzed by comparing worst-case response times of jobs with their timing constraints. We compute response times with a new method, which uses the amount of received service time to determine the response time of instances of a job. We illustrate how this method can be applied to exactly determine worst-case response times for processors with preemptive static-priority schedulers, and how it gives a good approximation on the response times for processors with non-preemptive static-priority scheduling or first-come-first-served scheduling. Our schedulability analysis method is the first to support systems with arbitrary job arrival patterns. Nevertheless, it performs better than other known approaches used for systems with periodic job arrivals.*

## 1. Introduction

In a distributed real-time system, jobs have stringent timing constraints and often require to be executed on a sequence of processors. Timing constraints are typically given in form of end-to-end deadlines. A job in such a system meets its timing constraint if it completes before its end-to-end deadline. If one or more jobs miss their deadlines, a timing failure occurs. The possibility of such timing failures makes the system difficult to validate, since their occurrence can have unexpected effects. Furthermore, timing failures can cause the system to behave in an unpredictable or unstable way, with potentially serious consequences. It is a design goal to guarantee a priori that timing requirements are met during the system's operation. If the job set in the system is static, design-time analysis validates that no timing constraints are violated in the system. If the job set is dynamic, additional run-time analysis, typically as part of

an admission control system, may be required.

The system workload is typically modeled as a set of job with end-to-end timing requirements. If all jobs in the system can meet their timing requirements, we call the system *schedulable*. The validation step required to test whether a system is schedulable is called *schedulability analysis*. The schedulability analysis can be performed during design time or as part of the admission control. In addition to being efficient, a method for schedulability analysis must satisfy a number of requirements. First, a schedulability analysis methodology must be correct and robust: it must never wrongly determine that a system is schedulable if it is not, and the error should be bounded if the schedulability decision is based on incorrect data. Second, a good schedulability analysis methodology should make good use of existing resources in the system, and allow for high resource utilization.

The central component of every schedulability analysis methodology is the computation of worst-case response times of the jobs under consideration. Once the worst-case response time has been determined for a particular job, it is compared against the timing requirements to check whether they are met<sup>1</sup>. In this paper, we present a general methodology for computing worst-case end-to-end response times for aperiodic jobs in distributed systems.

Traditionally, work on schedulability analysis focuses on periodic jobs, where the inter-arrival time of requests is fixed to be the period of the job. Non-periodic workload is typically transformed into a periodic workload by either one of three ways: (i) treating the non-periodic jobs as periodic jobs with the minimum inter-arrival time being the period, or (ii) having servers, which look like periodic jobs to the rest of

---

<sup>1</sup>Some approaches for schedulability analysis do not require the explicit computation of response times, but determine the schedulability indirectly, for example by relying on resource utilization [23].

the system, execute the non-periodic jobs (e.g. [16]), or (iii) splitting the non-periodic jobs each into collections of periodic jobs of different sizes and periods. In all three cases, well-known schedulability analysis methodologies for periodic workloads can be used.

Applying the same methods for distributed real-time systems, where jobs execute on more than one processor, shows poor results, even for periodic workloads. While the arrival of instances of a periodic job may indeed be periodic at the first processor, the completion of these instances almost certainly is not. If no special action is taken, and the completion of an instance on the first processor indicates that the second processor can go ahead, the "arrival" of instances of the job at the second processor is not periodic.

By appropriately synchronizing the execution of the job on the first processor and the start of the job on the second processor, the execution of the job on the second processor may be made to look like a periodic job. In [1], a number of such synchronization schemes are described and their relative performance is compared. The advantage of these synchronization schemes is that they allow the use of traditional schedulability analysis methods for periodic workloads.

As was pointed out in [1], appropriate synchronization reduces the worst-case end-to-end response times as compared to systems with no such synchronization (in [1] this is called Direct Synchronization). However, it adds overhead to the system, and increases the average end-to-end response times for jobs. In addition, it is of limited applicability in systems with jobs that are inherently aperiodic. The theory presented in this paper is designed to analyze aperiodic workloads. As such, it can handle periodic and aperiodic jobs, and combinations thereof, and more accurately determines the schedulability of periodic jobs in distributed systems with no synchronization than other approaches, for example [1].

## 2. Previous Work

The first result on schedulability analysis was presented in [23]. This schedulability test was performed by giving a utilization bound if the total utilization of the single processor is less than 69%, the rate monotonic scheduling will guarantee that all jobs meet their deadlines.

Since then, the results of [23] have greatly been generalized. For example, Lehoczky, Sha, and Ding [12] provide a sufficient and necessary schedulability test to determine the worst case response time. Leung and Whitehead [22] formulate an alternative priority assignment policy, where the job deadline can be less than the period of a job, and provide simple algorithm to determine the schedulability of such jobs. Sha, Rajkumar, and Lehoczky [14] discover a concurrency control protocol to permit jobs to share critical sections of codes. Audsley, Burns, Richardson, and

Welling [8] permit the addition of guaranteed sporadic tasks (where there is a minimum time between the re-arrivals of such jobs). Tindell, Burns, Richardson, Tindell, and Welling [9] extended the approach further to characterize the re-arrival pattern, covering 'bursty' sporadic and periodic jobs, and introduced the concept of release jitter (where a task is not released into the system immediately upon arrival, but may suffer a bounded deferral time). Bettati [4] provides a method for end-to-end schedulability analysis for distributed system. This approach relies on a synchronization scheme between processors called *Phase Modification*. Once an instance of a job completes on a processor, the release of the corresponding instance on the next processor is delayed so that the arrivals of that job on the second processor are periodic.

In [1, 2] Sun and Liu compare various synchronization mechanisms and describe an iterative algorithm to bound the end-to-end response times of jobs in distributed systems with *Direct Synchronization*. Direct synchronization between two processors means that the completion of an instance of a job on the first processor signals that the correspondent instance can be immediately released on the second processor. Sun and Liu correct a weakness in the holistic schedulability analysis proposed in [6]. However the upper bounds obtained by using their algorithm are still rather loose.

Most of the above work relies on one key technique, *busy period analysis*, which was first proposed in [13] and later extended in [6, 7, 9, 10]. A  $k$ -level busy period of a processor is a continuous time interval during which only these instances of jobs with priorities higher than or equal to  $k$  are executed. The crucial step of the busy period analysis can be roughly drafted as following: given a set of periodic jobs at the processor, for a particular job  $T_i$  with priority  $k$ , the maximum number of instances of job  $T_i$ , which arrive during  $k$ -level busy period  $D_k$ , can be bounded by  $\lceil \frac{D_k}{\rho_i} \rceil$ , where  $\rho_i$  is the period of job  $T_i$ , then the upper bound of the response time of each instance of job  $T_i$  can be obtained by only considering the first  $\lceil \frac{D_k}{\rho_i} \rceil$  instances of job  $T_i$ .

Unfortunately, busy period analysis in this form relies on jobs being periodic, and these schedulability analysis algorithms based on it are not applicable for job sets with bursty job arrivals.

## 3. System Model

In this section we describe the model for distributed real-time systems used in the following sections.

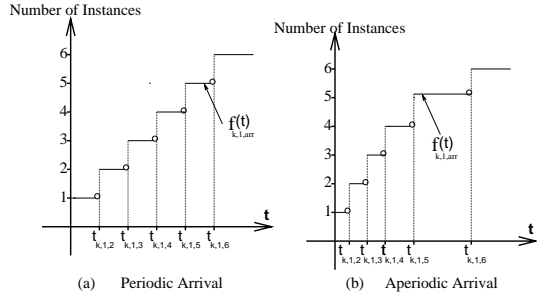
### 3.1. Jobs

We consider a distributed real-time system that consists of  $m$  processors  $P_1, P_2, \dots, P_m$  and  $n$  independent jobs  $T_1, T_2, \dots, T_n$ . Each job  $T_k$  consists of a chain of  $n_k$  subjobs,  $T_{k,1}, \dots, T_{k,n_k}$ . Subjobs of a job are executed on different processors sequentially. In particular,

subjob  $T_{k,j}$  is executed for  $\tau_{k,j}$  time units on processor  $P(k,j) \in \{P_1, P_2, \dots, P_m\}$ . We call  $\tau_{k,j}$  the *execution time* of subjob  $T_{k,j}$ .

Each job consists of a (possible infinite) sequence of job instances. The *release time* of an instance of a (sub-)job is the time when the instance of the (sub-)job is ready to be executed. Hence, by the definitions of job and subjob, the release time of an instance of job  $T_k$  is equal to the release time of the corresponding instance of subjob  $T_{k,1}$ . We say that the  $i$ -th instance of subjob  $T_{k,j}$  is released at time  $t_{k,j,i}$ . Naturally, we have  $0 \leq t_{k,j,1} < t_{k,j,2} < \dots < t_{k,j,i} < \dots$ .

Most previous studies assume that jobs are *periodic*. That is, the release time of the  $i$ -th instance of  $T_k$  follows the following relations:  $t_{k,1,i} = t_{k,1,1} + (i-1) * \rho_k$ , where  $\rho_k$  is the period of job  $T_k$ . In this study, we remove this assumption. We allow that instances of jobs are released at any point in time, not necessarily periodically. Examples for both periodic and aperiodic job arrivals are given in Figure 1. Each job  $T_k$  is associated with a deadline  $D_k$ .



**Figure 1. Arrival functions of the first subjob.**

For each instance of job  $T_k$ , the end-to-end response time (the time between the release time of the first subjob and completion time of the last subjob) must be no more than the deadline.

### 3.2. Scheduling algorithms

A processor typically executes more than one subjob. Hence, a scheduler is needed in order to coordinate the executions of subjobs on the processor. Priority scheduling is widely used. With priority scheduling, subjobs waiting for execution on a processor are assigned priorities. Among all the ready subjobs, the scheduler picks the one with the highest priority. A priority scheduling algorithm is *static* if all instances of a subjob have the same priority. Static priority scheduling is easy to manage and implement. A priority scheduling algorithm is *preemptive* if it preempts the current subjob in order to run an instance of a higher-priority subjob which just arrives. In this paper, we will consider both *static priority preemptive* (SPP) and *static priority non preemptive* (SPNP) scheduling algorithms. Given that SPP or SPNP scheduling is used, the response time of jobs is sensitive to how priorities are assigned to subjobs. Priority assignment algorithms have been widely studied in [9, 22, 23], and are not further discussed in this paper, since our results apply to

arbitrary priority assignments. In the following we assume that subjob is assigned priority  $\phi_{k,j}$  on processor  $P(k,j)$ . The smaller the value of  $\phi_{k,j}$ , the higher the priority of  $T_{k,j}$ .

In addition to static priority scheduling, we analyze first-come-first-served (FCFS) scheduling, where instances of subjobs are served in accordance to the order of their arrivals.

As described earlier, we do not enforce a particular synchronization scheme except that dependency constraints between subsequent subjobs must be maintained. We therefore assume that the completion of an subjob on one processor signals that the corresponding instance of the next subjob is released on the subsequent processor. This assumption can be enforced with the help of the Direct Synchronization Protocol [1] to signal the completion of a subjob. When an instance of a subjob completes processing, a synchronization signal is sent to the processor where its immediate successor executes. Consequently, an instance of its successor is released immediately. This protocol is easy to use and implement. We also assume that the scheduler overhead has been taken into account in the execution time of the subjob. The inter-processor communication overhead is assumed to be constant and, hence, is ignored.

## 4. Response Time Analysis

To determine the worst case end-to-end response times of jobs, we need to describe job arrivals and departures to and from processors, the time requested by subjobs from a particular processor, and the time offered by processors to a particular job. We define the following notations for this purpose.

**Definition 1** The arrival function,  $f_{k,j,arr}(t)$ , of subjob  $T_{k,j}$  is defined as the number of instances of subjob  $T_{k,j}$  that are released during the time interval  $[0, t]$ .

Obviously, the value of the *arrival function* increases at every release time of a subjob instance. In particular, we have  $f_{k,j,arr}(t) = i$  for  $t_{k,j,i} \leq t < t_{k,j,i+1}$ .

**Definition 2** The departure function,  $f_{k,j,dep}(t)$ , of subjob  $T_{k,j}$  is defined as the number of instances of subjob  $T_{k,j}$  that are completed during time interval  $[0, t]$ .

Since jobs become ready on a succeeding processor as soon as they complete on the current processor, we always have  $f_{k,j,dep}(t) = f_{k,j+1,arr}(t)$ . In particular, we have  $f_{k,j,dep}(t) = i$  for  $t_{k,j+1,i} \leq t < t_{k,j+1,i+1}$ .

**Definition 3** The workload function of subjob  $T_{k,j}$  is defined as

$$c_{k,j}(t) = f_{k,j,arr}(t) * \tau_{k,j}, \quad (1)$$

where  $\tau_{k,j}$  is execution time of subjob  $T_{k,j}$ .

**Definition 4** The service function,  $S_{k,j}(t)$ , of processor  $P(k,j)$  for subjob  $T_{k,j}$  is defined as the time of processor  $P(k,j)$  taken to execute ready instances (if any) of subjob  $T_{k,j}$  during the time interval  $[0, t]$ .

These arrival, departure, workload, and service functions play a key role when deriving the end-to-end response times of jobs. Before we formally derive our main results, we need to introduce a few more mathematical notations.

**Definition 5** For a nondecreasing function  $g(t)$ , the inverse function of  $g(t)$  is defined as

$$g^{-1}(t) = \min\{s | g(s) \geq t\}. \quad (2)$$

For example, the inverse function of the arrival function  $f_{k,j, \text{arr}}(t)$  can be written as follows: for  $m = 1, 2, \dots$

$$f_{k,j, \text{arr}}^{-1}(m) = t_{k,j,m}. \quad (3)$$

That is, while  $f_{k,j, \text{arr}}(t)$  denotes the numbers of instances of subjob  $T_{k,j}$  released during time interval  $[0, t]$ ,  $f_{k,j, \text{arr}}^{-1}(m)$  is the time when the  $m$ -th instance of subjob  $T_{k,j}$  is released.

**Definition 6** A function  $\underline{g}(t)$  (or  $\overline{g}(t)$ ) is called a lower bound function (or an upper bound function) of function  $g(t)$  if for  $t \geq 0$ ,

$$g(t) \geq \underline{g}(t) \quad (\text{or } g(t) \leq \overline{g}(t)). \quad (4)$$

For example,  $t$  is an upper bound of  $S_{k,j}(t)$  and 0 is a lower bound of  $S_{k,j}(t)$ . Hence, we have

$$\overline{S}_{k,j}(t) = t, \quad (5)$$

and

$$\underline{S}_{k,j}(t) = 0. \quad (6)$$

Of course, these bounds are very loose. As we will see below, the quantity of the response time bounds directly depends on whether tight upper and lower bounds on service functions can be found.

## 4.1. Exact analysis

### 4.1.1 Main results

The following theorem provides a fundamental formula for computing the exact value of worst case end-to-end response times.

**Theorem 1** The worst case end-to-end response time  $d_k$  of job  $T_k$  is given as follows:

$$d_k = \max_{m \geq 0} (f_{k,n_k}^{-1} \text{dep}(m) - f_{k,1}^{-1} \text{arr}(m)). \quad (7)$$

**Proof:** Due to space limitation, all proofs in this paper are omitted. An interested reader is referred to [18]. Q.E.D

Typically for real-time systems we may assume that the arrival functions of the first subjobs are known. Hence, we need to determine the departure function on the last processor in order to use Formula (7). The following theorem establishes a relationship between service function and departure function on a single processor.

**Theorem 2** Let  $S_{k,j}(t)$  be the service function for subjob  $T_{k,j}$  at processor  $P(k,j)$ . Then, its departure function  $f_{k,j, \text{dep}}(t)$ , is given by

$$f_{k,j, \text{dep}}(t) = \lfloor \frac{S_{k,j}(t)}{\tau_{k,j}} \rfloor. \quad (8)$$

### 4.1.2 Service Functions for SPP Scheduling

As described above, we need to derive the service function in order to use Formula (7) to compute the end-to-end response time of a job. Service functions depend on the scheduling algorithm used by processors. In general, the derivation of exact service function is difficult. For a number of scheduling algorithms it can be derived rather easily, however. The following theorem illustrates this for the case of static priority preemptive (SPP) scheduling.

**Theorem 3** The service function  $S_{k,j}(t)$  for subjob  $T_{k,j}$  on processor  $P(k,j)$  that uses SPP scheduling is given by

$$S_{k,j}(t) = \min_{0 \leq s \leq t} \{A_{k,j}(t) - A_{k,j}(s) + c_{k,j}(s)\}, \quad (9)$$

where

$$A_{k,j}(t) = \begin{cases} t, & \phi_{k,j} = 1 \\ t - \sum_{P(h,i)=P(k,j), \phi_{h,i} < \phi_{k,j}} S_{h,i}(t), & \phi_{k,j} > 1 \end{cases} \quad (10)$$

Equations (9) (10) illustrates that the service function of  $T_{k,j}$  depends on two items: (i) service functions of higher-priority subjobs that are also executed on processor  $P(k,j)$  and (ii) the workload function of  $T_{k,j}$ , which in turn depends on the arrival function of  $T_{k,j}$ . Thus, the service function of  $T_{k,j}$  can be obtained by first computing all the service functions of higher priority subjobs and the service function at predecessor processor. Once the service function is computed, we can obtain the departure function with the help of Formula (8). The departure function in turn is the arrival function on the subsequent processor. Substituting the departure function on the last processor  $P(k, n_k)$  into Formula (7), we have the worst case end-to-end response time of  $T_k$ .

## 4.2. Approximate Analysis

In order to use Formula (7) directly, one must be able to accurately compute departure functions at every processor. For many scheduling algorithms, this is either too difficult or computationally very intensive. In this situation, we have to use approximation techniques. We address this problem in this subsection.

### 4.2.1 Main results

According to the following theorem, we see that if the departure functions can be lower bounded and the arrival functions can be upper bounded, then the worst case end-to-end response time can be upper bounded.

**Theorem 4** If  $\overline{f}_{k,j, \text{arr}}(t)$  and  $\underline{f}_{k,j, \text{dep}}(t)$  are known for all the subjobs of job  $T_k$ , its worst case end-to-end response time  $d_k$  can be approximated by

$$d_k \leq \sum_{j=1}^{n_k} d_{k,j}, \quad (11)$$

where  $d_{k,j}$  is given by

$$d_{k,j} = \max_{m \geq 0} (\underline{f}_{k,j, \text{dep}}^{-1}(m) - \overline{f}_{k,j, \text{arr}}(m)). \quad (12)$$

Given the above theorem, in order to compute an upper bound of the worst case response time, we need to estimate the lower bound of the *departure function* and the upper bound of the *arrival function*. The following lemmas relate these bounds to those of *service functions*.

**Lemma 1** A lower bound function on the departure function  $f_{k,j,dep}(t)$  of subjob  $T_{k,j}$  on processor  $P(k,j)$  is given by

$$f_{k,j,dep}(t) = \lfloor \frac{\underline{S}_{k,j}(t)}{\tau_{k,j}} \rfloor. \quad (13)$$

**Lemma 2** An upper bound on the arrival function  $f_{k,j+1,arr}(t)$  of subjob  $T_{k,j+1}$  on processor  $P(k,j+1)$  is given by

$$\bar{f}_{k,j+1,arr}(t) = \lfloor \frac{\bar{S}_{k,j}(t)}{\tau_{k,j}} \rfloor. \quad (14)$$

The question is how to obtain the upper and lower bounds of the service function. We address this problem in the next section for the special cases of static priority non-preemptive scheduling (SPNP), and the first-come-first-served scheduling (FCFS).

#### 4.2.2 Bounds on Service Functions for SPNP Scheduling

Recall that in a processor that uses non-preemptive static priority scheduling once a subjob begins to execute, it cannot be interrupted, even if higher priority subjobs subsequently arrive. Lower priority jobs thus can temporarily prevent higher priority jobs from executing. In such a situation, the higher priority subjob is said to be *blocked* by the lower priority subjobs. This blocking complicates the response time computation.

The maximum blocking time  $b_{k,j}$  of subjob  $T_{k,j}$  is the maximum execution time of subjobs that are assigned lower priority than subjob  $T_{k,j}$  on processor  $P(k,j)$ . Formally,

$$b_{k,j} = \max_{P(l,m)=P(k,j), \phi_{l,m} > \phi_{k,j}} \{\tau_{l,m}\}. \quad (15)$$

Once the blocking time is known, we can estimate the bounds on *service functions* as described in the following theorems.

**Theorem 5** A lower bound function on the service function  $S_{k,j}(t)$  of subjob  $T_{k,j}$  on processor  $P(k,j)$ , which uses static priority non-preemptive scheduling, is given by

$$\underline{S}_{k,j}(t) = \begin{cases} 0, & t \leq b_{k,j} \\ \min_{0 \leq s \leq t - b_{k,j}} \{B_{k,j}(t) - B_{k,j}(s) + c_{k,j}(s)\}, & t > b_{k,j} \end{cases} \quad (16)$$

where

$$B_{k,j}(t) = \begin{cases} 0, & t \leq b_{k,j} \\ t - b_{k,j}, & t > b_{k,j}, \phi_{k,j} = 1 \\ t - b_{k,j} - \sum_{P(h,i)=P(k,j), \phi_{h,i} < \phi_{k,j}} \underline{S}_{h,i}(t), & t > b_{k,j}, \phi_{k,j} > 1 \end{cases} \quad (17)$$

**Theorem 6** An upper bound function service function on the service function  $S_{k,j}(t)$  on processor  $P(k,j)$ , which uses static priority non-preemptive scheduling, is given by

$$\bar{S}_{k,j}(t) = \min_{0 \leq s \leq t} \{B_{k,j}(t) - B_{k,j}(s) + c_{k,j}(s)\}, \quad (18)$$

where  $\underline{S}_{h,i}(t)$  is defined in Theorem 5 and

$$B_{k,j}(t) = \begin{cases} t, & \phi_{k,j} = 1 \\ t - \sum_{P(h,i)=P(k,j), \phi_{h,i} < \phi_{k,j}} \underline{S}_{h,i}(t), & \phi_{k,j} > 1 \end{cases} \quad (19)$$

Thus, with the above theorems, the lower and upper bounds of service functions can be obtained. These bounds can be substituted into Equations (13) and (14) to derive lower and upper bounds on *departure* and *arrival functions*, respectively. These are then substituted into Equation (12) to determine a bound on the local response time for a single subjob. The bound on the end-to-end response time is then determined as the sum of local response times for all the subjobs.

#### 4.2.3 Bounds on service functions for FCFS scheduling

In order to estimate *service functions*, we need to know how much time offered to execute subjobs in time interval  $[0, t]$  by the processor. We derive this with the notation of the *utilization function* defined as follows:

**Definition 7** The utilization function  $U_j(t)$  of processor  $P_j$  is defined as the time processor  $P_j$  is busy executing subjobs during the time interval  $[0, t]$ .

Obviously,  $U_j(t)$  can not exceed  $t$ . If  $U_j(t) = t$ ,  $t \in [0, T]$ , processor  $P_j$  is busy during the entire time interval  $[0, T]$ . If  $U_j(t) < t$ , processor  $P_j$  must be idle for some time before time  $t$ . So  $U_j(t)$  can be seen as an indicator of how busy processor  $P_j$  is.

**Theorem 7** The utilization function  $U_j(t)$  of processor  $P_j$ , for the case of FCFS scheduling, is given by

$$U_j(t) = \min_{0 \leq s \leq t} \{t - s + G_j(s)\}, \quad (20)$$

where,

$$G_j(t) = \sum_{P(k,l)=P_j} c_{k,l}(t). \quad (21)$$

While the FCFS algorithm seems to be a simple one, analyzing it in order to obtain the service function is not trivial. This is because, with FCFS scheduling, a processor arbitrarily picks up a subjob to execute from more than one subjobs if they arrive at the same time. Thus, it is difficult, if not impossible, to obtain the exact *service function* for a subjob executed on a processor using FCFS scheduling. Nevertheless, the following theorems provide upper and lower bounds on the service functions when using FCFS scheduling. In the following, we will have  $U_{k,j}(t)$  denote the utilization function of processor  $P(k,j)$ . Similarly,  $G_{k,j}(t)$  denotes the total workload of all subjobs on processor  $P(k,j)$ .

**Theorem 8** If  $P(k, j)$  uses the FCFS scheduling algorithm, the service function  $S_{k,j}(t)$  for subjob  $T_{k,j}$  is lower bounded by

$$\underline{S}_{k,j}(t) = c_{k,j}(G_{k,j}^{-1}(U_{k,j}(t))), \quad (22)$$

where  $G_{k,j}(t)$  and  $U_{k,j}(t)$  are defined in Theorem 7.

**Theorem 9** If  $P(k, j)$  uses the FCFS scheduling algorithm, the service function  $S_{k,j}(t)$  of subjob  $T_{k,j}$  is upper bounded by

$$\overline{S}_{k,j}(t) = c_{k,j}(G_{k,j}^{-1}(U_{k,j}(t))) + \tau_{k,j}. \quad (23)$$

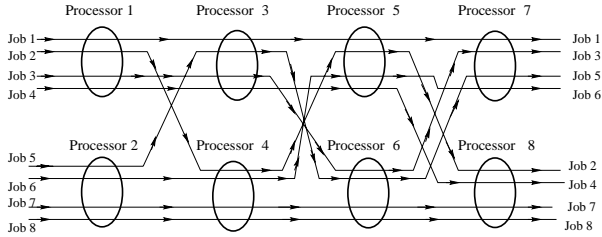
where  $G_{k,j}(t)$  and  $U_{k,j}(t)$  are given in (21) and (20), respectively.

As in the case of the static priority non-preemptive scheduling (Section 4.2.2), once the lower and upper bounds on service functions are obtained, an upper bound on the worst case end-to-end response time can be computed by using (13), (14), (12), and (11).

## 5. Evaluation

We conducted a series of simulations to study the performance of the proposed methods for analysis of response time in the distributed real-time systems with various scheduling algorithms. We are going to demonstrate that our new method generates tighter bounds on response time than approaches proposed by others [1, 2], for both the case of periodic and aperiodic job arrivals.

### 5.1. Simulation model and assumptions



**Figure 2. A System with Four Stages**

In our experiments, we simulate the execution of jobs in a job shop. The shop consists of a sequence of stages, each of which contains a number of processors. All jobs traverse the stages of the shop in the same order, and each job is assigned to execute on one processor in each stage. Figure 2 shows a shop configuration, which consists of four stages with two processor in each stage. For example, job  $T_1$  is assigned to execute on  $P_1$  in the first stage, and on  $P_3$ ,  $P_5$  and  $P_7$  in the second, third and fourth stage, respectively. Job  $T_2$  executes on  $P_1$  in the first stage, and on  $P_4$ ,  $P_6$  and  $P_8$  on the subsequent stages.

In the case of static priority scheduling, the priority assignment must be determined. We use a relative deadline monotonic priority assignment algorithm [1], which assigns

priorities to subjobs as follows: First, a sub-deadline of subjob  $T_{i,j}$  is defined as follows

$$D_{i,j} = \frac{\tau_{i,j}}{\sum_{k=1}^{n_i} \tau_{i,k}} D_i. \quad (24)$$

Then, subjobs on a particular processor are assigned priorities in accordance to their sub-deadlines. The smaller the sub-deadline of a subjob, the higher its priority.

We simulate four different methods for obtaining worst case end-to-end response times:

- SPP/Exact: The exact analysis method for static priority preemptive scheduling as proposed in Section 4.1.
- SPNP/App: The approximate method for static priority non preemptive scheduling (SPNP) as proposed in Section 4.2.2.
- FCFS/App: The approximate method for FCFS scheduling as proposed in Section 4.2.3
- SPP/S&L: The method proposed in [1, 2]. This method is associated with static priority preemptive scheduling.

We measure the performance of each scheme in terms of admission probability. The *admission probability* is defined as the probability that a randomly generated job set can meet its deadline requirements. We are interested in measuring how the different analysis methods perform with different scheduling algorithms. In each run of the simulation, 1,000 sets of jobs are randomly generated. We apply each analysis method separately to determine how many sets of jobs can be admitted (i.e., meet their deadline requirements). The admission probability is estimated by the percentage of job sets that are admitted. Separate simulation runs are made to measure the admission probability when job arrivals are periodic and aperiodic.

### 5.2. Numerical results

The results of our experiments with periodic and aperiodic jobs arrivals are presented in Figure 3 and Figure 4, respectively.

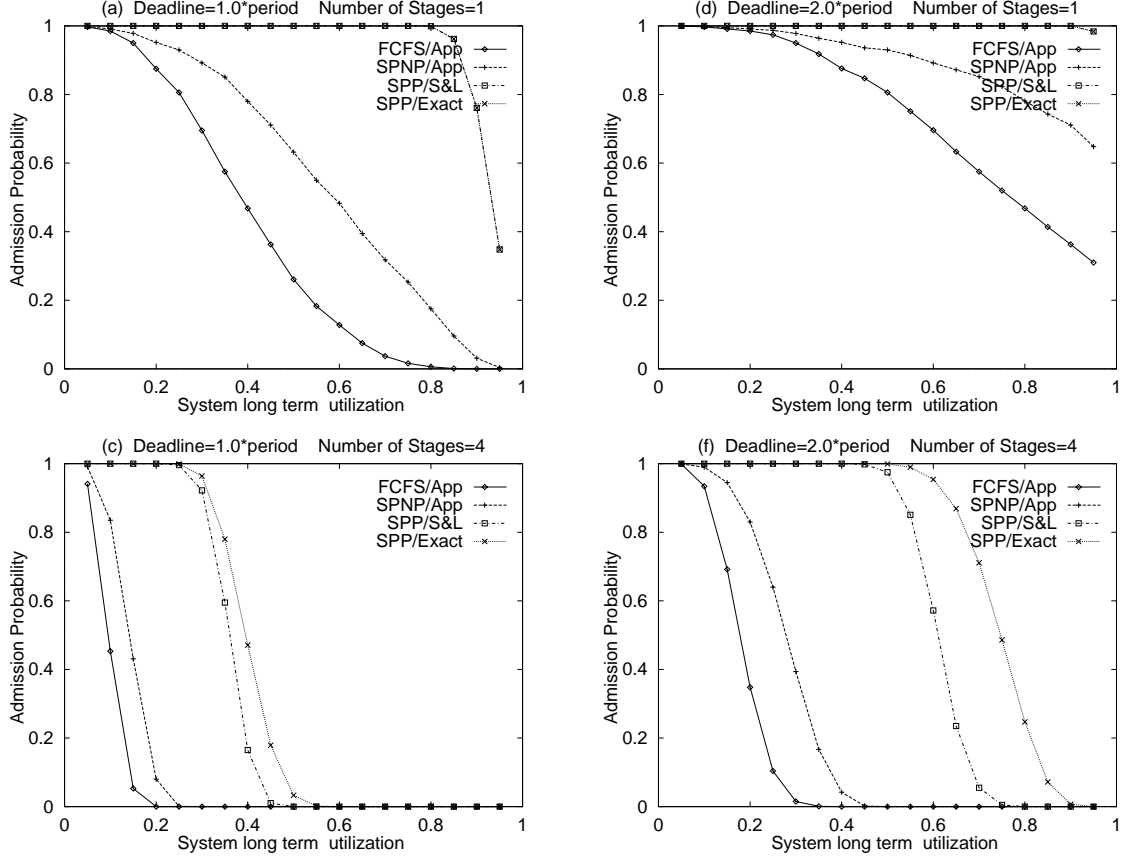
In Figure 3 we compare the admission probability of the four analysis methods for the case of periodic job arrival. For each job  $T_k$ , we use following formula to generate release times of the first subjob  $T_{k,1}$ : for  $m = 1, 2, \dots$ ,

$$t_{k,1,m} = \frac{m-1}{x_k}, \quad (25)$$

where  $x_k$  is a random variable with uniform distribution in  $(0, 1)$ . The end-to-end deadline of job  $T_k$  is a multiple of the period  $\frac{1}{x_k}$ . Furthermore, we generate a random variable  $w_{k,j}$  with uniform distribution in  $(0, 1)$  for each subjob  $T_{k,j}$  and the execution time  $\tau_{k,j}$  of subjob  $T_{k,j}$  is defined as:

$$\tau_{k,j} = \frac{w_{k,j} * \frac{1}{x_k}}{\sum_{P(l,i)=P(k,j)} w_{l,i} * \frac{1}{x_l}} * Utilization. \quad (26)$$

Figure 3 shows the effects of increasing the number of stages in the shop (from top to bottom) and of increasing



**Figure 3. Admission Probability vs. System Utilization for Periodic Arrival Pattern.**

the end-to-end deadlines of jobs (from left to right). We note that both FCFS/App and SPNP/App perform consistently poorer than the other two approaches. This is only partly due to the approximate analysis. Preemptive static priority scheduling is inherently superior to non-preemptive static priority scheduling or to FCFS, independently of the analysis methodology. More interesting are the results for SPP/Exact and SPP/S&L, which compare two different analysis methodologies for identical systems, with identical scheduling algorithm. When the number of stages is one (i.e., Figure 3 (a) and (d)), both systems using SPP/Exact and SPP/S&L result in the identical performance. This means that for a single processor system, both methods predict the same response time. However, when the number of stages is more than one (i.e., Figure 3 (c) and (f)), SPP/Exact performs better. This is because our SPP/Exact is an exact analysis method, which accurately compute the end-to-end response time, while SPP/S&L implicitly overestimates the subjob arrivals and result in a loose bound on end-to-end response time. As to be expected the performance of all four methods improves significantly as the end-to-end deadline is doubled.

Figure 4 compares the performance of SPP/Exact,

SPNP/App, and FCFS/App for aperiodic job arrivals. In these experiments, we do not compare with SPP/S&L, because their analysis method works for periodic job arrivals only. For each job  $T_k$ , we use following formula to generate release times of the first subjob  $T_{k,1}$ : for  $m = 1, 2, \dots$ ,

$$t_{k,1,m} = \frac{1}{x_k} \sqrt{x_k^2 + (m-1)^2} - 1, \quad (27)$$

where  $x_k$  is a random variable with uniform distribution in  $(0, 1)$ . The end-to-end deadline of job  $T_k$  is a random variable with exponential distribution. Furthermore, we generate a random variable  $w_{k,j}$  with uniform distribution in  $(0, 1)$  for each subjob  $T_{k,j}$  and the execution time  $\tau_{k,j}$  of subjob  $T_{k,j}$  is defined as:

$$\tau_{k,j} = \frac{w_{k,j} * \frac{1}{x_k}}{\sum_{P(l,i)=P(k,j)} w_{l,i} * \frac{1}{x_l}} * Utilization. \quad (28)$$

Figure 4 shows how the performance of the three methods compare with varying deadline distributions, from top to bottom the variance of the distribution increases, while the average value increases from left to right. As expected, performance improves as the deadlines are larger, as there are more slack in the systems. The figure shows, however,

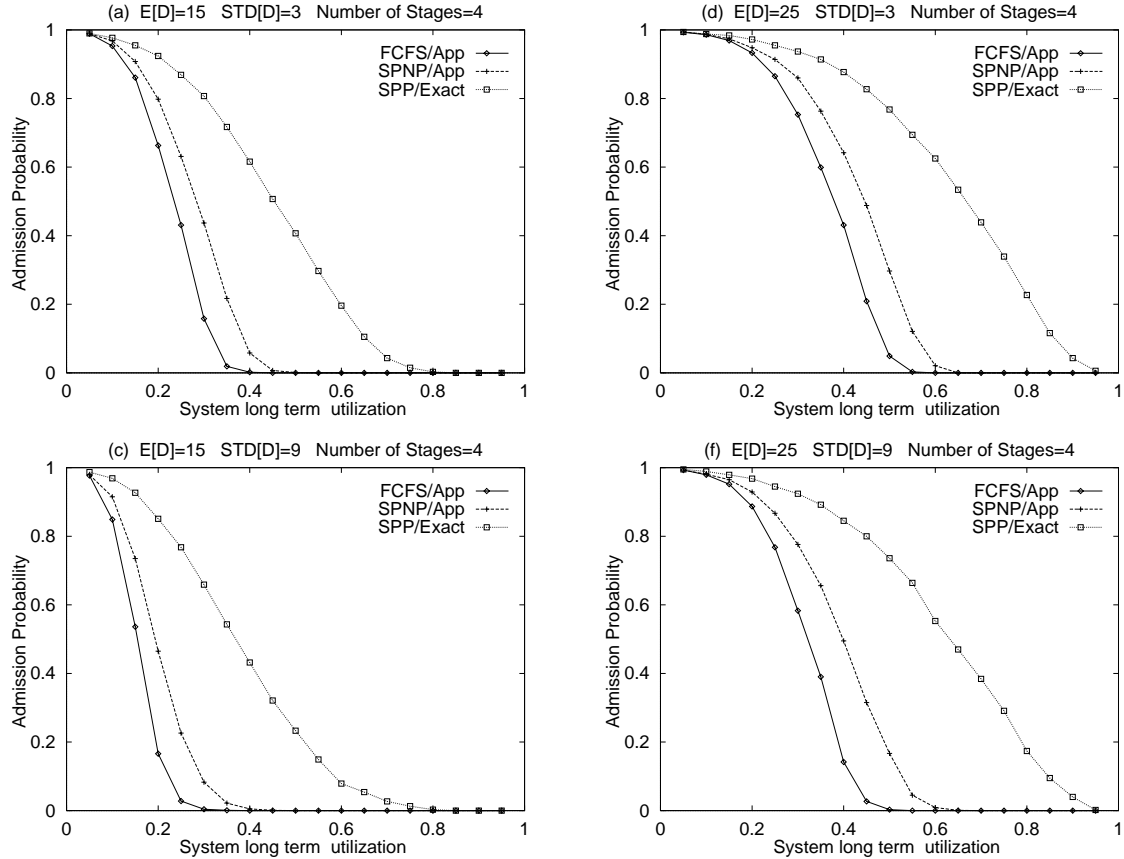


Figure 4. Admission Probability vs. System Utilization for Aperiodic Arrival Pattern.

that changing the variance of deadlines has a little effect on the admission probability.

The results show here are obtained for a limited number of parameters. However, we also found that other parameter values led to similar observations and are not represented here due to the lack of space.

## 6. Conclusion

In this paper, we have presented a novel approach to schedulability analysis of distributed real-time systems that have arbitrary job arrival patterns, and that consist of processors that run preemptive and non-preemptive static priority schedulers and FCFS schedulers. The basis of our new theory is the development of formulae that use the service received by the job from the processor and the service required by the job from the processor to bound the worst case response time of the job on that processor.

This paper makes a number of contributions. First, it allows the schedulability analysis for very general aperiodic workloads. Second, we have shown how this methodology can be used for systems that have a variety of different schedulers, be they static-priority (preemptive or non-preemptive) or FCFS. Of course the proposed methodology can handle

heterogeneous systems, where different processors run different schedulers. Third, we have shown that our approach gives good results for systems with periodic job arrivals as well, in particular in comparison with recently developed methods, such as [1, 2].

A number of questions remain open. We are investigating more general methodology to deal with the "physical loop" caused by jobs  $\Phi$  visiting the same processor more than once and the "logical loop" caused by certain jobs disturbing each other on different processors. In these situations, the arrival functions of some subjobs, which play the key in computing the worst case response time, depend on each other and form a closed relationship chain. For example, the arrival function of subjob  $T_{k,j}$  depends on the arrival function of subjob  $T_{n,i}$ , because subjob  $T_{k,j-1}$  and subjob  $T_{n,i}$  are served by the same processor and the priority of subjob  $T_{n,i}$  is higher than that of subjob  $T_{k,j-1}$ . Furthermore, the arrival function of subjob  $T_{n,i-1}$  depends on the arrival function of subjob  $T_{k,j}$ , because subjob  $T_{n,i-1}$  and subjob  $T_{k,j}$  are served by the same processor and the priority of subjob  $T_{k,j}$  is higher than that of subjob  $T_{n,i-1}$ . In order to evaluate the worst case response time, we need to virtually break the closed chain. Observing that an upper

bound of arrival function of each subjob can be obtained from the arrival function of its precursory subjob and the worst case response time experienced by the its precursory subjob. Let the worse case response time of each subjob be an unknown variable. According to above observation, we can obtain the upper bounds of arrival functions for each subjob even though they may contain some unknown parameters. From the mathematical point of view, after setting the worse case response time as unknown variable, we can construct a nonlinear vector function,  $\vec{X} = F(\vec{X})$ , where  $\vec{X}$  is the unknown vector consisting of all unknown worst case response time experienced by each subjob. Therefore, we can find the numerical solution  $\vec{X}$  by using the iteration scheme:  $\vec{X}^{n+1} = F(\vec{X}^n)$ ,  $\vec{X}^1 = \vec{0}$ ,  $n = 1, 2, \dots$ .

In this paper we restricted ourselves to distributed systems with no contention for resources, except for the processors. Large-scale distributed systems cannot be modeled without taking into account remote resource access and contention for such resources. We are currently investigating how to model the access to shared resources (with and without resource access protocols) with the help of service functions. This will open the way for a fully integrated methodology for the schedulability analysis of distributed real-time systems with both shared processors and shared resources.

## Acknowledgment

This work was partially sponsored by the Air Force Office of Scientific Research, Air Force Material Command, USAF, under grant number F49620-96-1-1076, by the Defense Advanced Research Projects Agency (DARPA) through the Honeywell Technology Center under contract number B09333438, and by Texas Higher Education Coordinating Board under its Advanced Technology Program with grant number 999903-204. The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Air Force Office of Scientific Research, Honeywell, DARPA, the U.S. Government, Texas State Government, Texas Higher Education Coordinating Board, or Texas A&M University.

## References

- [1] J. Sun and J. W. S. Liu. Synchronization protocols in distributed real-time systems. In *Proc. of IEEE ICDCS*, 1996.
- [2] J. Sun and J. W. S. Liu. Bounding the end-to-end response times of tasks in a distributed real-time system using the direct synchronization protocol. Technical report UIUCDCS-R-96-1949, University of Illinois at Urbana-Champaign, Dept. of Computer Science, March 1996.
- [3] J. Sun. *Fixed-Priority Scheduling Of Periodic Tasks With End-To-End Deadlines*. PhD thesis, University of Illinois at Urbana-Champaign, 1996.
- [4] R. Bettati. *End-to-End Scheduling to Meet Deadlines in Distributed System*. PhD thesis, University of Illinois at Urbana-Champaign, 1994.
- [5] R. Bettati and J. W. S. Liu. End-to-End Scheduling to Meet Deadlines in Distributed Systems. In *Proc. of IEEE ICDCS*, 1992.
- [6] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time system. In *Microprocessing and Microprogramming*, 50(2), April 1994.
- [7] K. Tindell, A. Burns and A. J. Wellings. An extensible approach for analyzing fixed priority hard real-time tasks. In *J. of Real-Time Systems*, 6(2), March 1994.
- [8] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: the deadline-monotonic approach. In *Proc. of IEEE Workshop on Real-Time Operating Systems and Software*, 1991.
- [9] N. C. Audsley, A. Burns, M. F. Richardson, K. Tindell and A. J. Wellings. Applying new scheduling theory to static priority preemptive scheduling. In *J. of Software Eng.*, vol 8, no. 5, Sept. 1993
- [10] A. Burns, K. Tindell and A. J. Wellings. Fixed priority scheduling with deadlines prior to completion. In *Proc. of Euromicro Workshop on Real-Time Systems*, 1994.
- [11] R. I. Davis, K. Tindell and A. Burns. Scheduling slack time in fixed priority pre-emptive systems. In *Proc. of IEEE RTSS*, 1993.
- [12] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proc. of IEEE RTSS*, 1989.
- [13] J. Lehoczky. Fixed priority scheduling of periodic task sets with arbitrary deadlines. In *Proc. of IEEE RTSS*, Dec. 1990.
- [14] L. Sha, R. Rajkumar and J. Lehoczky. Priority inheritance protocols: an approach to real-time synchronization. In *IEEE Transactions on Computers*, vol. 39, no. 9, Sept. 1990.
- [15] J. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Proc. of IEEE RTSS*, 1992.
- [16] S. Ramos-Thuel and J. Lehoczky. On-line scheduling of hard deadline aperiodic tasks in fixed-priority systems. In *Proc. of IEEE RTSS*, 1993.
- [17] C. Li, R. Bettati and W. Zhao. Static priority scheduling for ATM networks. In *Proc. of IEEE RTSS*, 1997.
- [18] C. Li, R. Bettati and W. Zhao. Response time analysis for distributed real-time systems with bursty job arrivals. Technical report, Department of Computer Science, Texas A&M University, 1998.
- [19] M. Joseph and P. Pandya. Finding response times in a real-time system. In *Journal of Computerl*, vol. 29, no. 5, 1986.
- [20] R. L. Cruz. A calculus for network delay, part I,II: Network analysis. In *IEEE Trans. on Inform. Theory*, 37(1), Jan. 1991.
- [21] R. L. Cruz. Quality of service guarantees in virtual circuit switched networks. In *IEEE Journal on Selected Areas in Commu.*, vol. 13, no. 6, 1995.
- [22] J. Y.-T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. In *Performance Evaluation*, 2, December 1982.
- [23] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. In *J. of the Association for Computing Machinery*, 20(1), Jan. 1973.
- [24] J. A. Stankovic. Misconceptions about real-time computing: A serious problem for next generation systems. In *J. of IEEE Computer*, 21(10), Oct. 1988.
- [25] J. A. Stankovic and K. Ramamritham, editors. *Hard Real-Time Systems*. IEEE Computer Society Press, 1988.
- [26] Chang-Gun Lee, J. Hahn, Y. M. Seo, S. L. Min and R. Ha. Analysis of Cache-related Preemption Delay in Fixed-priority Preemptive Scheduling. In *Proc. of IEEE RTSS*, 1996.