The Case for Non-Preemptive Scheduling in Distributed Real-Time Systems

Praveen Jayachandran and Tarek Abdelzaher
Department of Computer Science
University of Illinois
Urbana-Champaign, IL 61801
e-mail: *pjayach2@uiuc.edu, zaher@uiuc.edu*

Abstract

Contrary to traditional belief, we show in this paper, that for distributed systems non-preemptive scheduling can perform better than preemptive scheduling in the worst case in terms of task schedulability, under certain circumstances. We derive a worst-case delay bound for tasks scheduled using non-preemptive scheduling in a distributed system, where the task flow paths form a directed acyclic graph. The delay bound leads to a reduction of the distributed system to an equivalent hypothetical uniprocessor system scheduled using preemptive scheduling. This transformation enables the wealth of uniprocessor schedulability theory to be applied to analyze schedulability under non-preemptive scheduling in distributed directed acyclic systems. Our simulation studies show that non-preemptive scheduling can perform better than preemptive scheduling for distributed systems. We also characterize through simulations the scenarios under which non-preemptive is better than preemptive scheduling, and scenarios where the opposite is true. We hope this paper will serve as a first step towards more extensive study and use of non-preemptive scheduling in distributed systems.

1. Introduction

Large distributed and embedded systems are becoming increasingly prevalent. Understanding the end-to-end temporal behavior and ensuring schedulability of such systems is a fundamental concern of real-time computing. While a plethora of schedulability analysis techniques addressed multiprocessors over the last decade, there is a distinct lack of theoretical tools to conduct such analysis on distributed systems.

In uniprocessor scheduling, it is well known that preemptive scheduling performs better than non-preemptive scheduling in terms of task schedulability, and the same is widely believed to be true for distributed systems too. This paper presents a case contrary to this traditional belief. By deriving a worst case delay bound for jobs under non-preemptive scheduling, we show that non-preemptive scheduling can ensure a better worst case performance than preemptive scheduling in distributed systems, under certain conditions.

In this paper, we are concerned with distributed systems that process several classes of real-time tasks, whose execution paths form a Directed Acyclic Graph (DAG). Each task traverses through multiple stages of execution and must exit the system within specified end-to-end latency bounds. Non-preemptive prioritized scheduling is assumed at each node of the DAG. We derive a *delay composition rule* that allows the worst-case delay of a task invocation to be expressed in terms of the execution times of other task invocations that are present in the system concurrently with it. We provide an intuition as to why this worst case estimate can be better than the worst case scenario for the same task set under preemptive scheduling. The simple expression of end-to-end delay bound computed by the aforementioned delay composition rule leads to a reduction of the multi-stage distributed system to an equivalent hypothetical uniprocessor system scheduled using preemptive scheduling. Using this transformation, it becomes possible to use a wealth of existing schedulability analysis techniques for preemptive scheduling on the new single-processor task set to analyze the original distributed system scheduled using non-preemptive scheduling. For this reason, we call this a 'meta-schedulability test'. Our evaluations through simulation studies show that the meta-schedulability test when used as an admission controller for non-preemptive scheduling in distributed systems can admit a much higher per-stage utilization while still meeting all task deadlines, compared to preemptive scheduling. Through simulation studies, we characterize the situations under which non-preemptive scheduling performs better than preemptive scheduling, and also the situations when the opposite is true.

Our delay composition rule does not make assumptions on the scheduling policy or the periodicity of the task set. No assumption is made on whether different invocations of the same task have the same priority. Hence, this rule applies to static-priority scheduling (such as rate-monotonic), dynamic-priority scheduling (such as EDF) and aperiodic task scheduling alike. The proposed reduction of the distributed system to an equivalent uniprocessor system, however, assumes that different invocations of the same task have the same priority at each stage, although the task can be as-

signed different priorities on different stages.

Schedulability theory over the last thirty years, has for the most part ignored non-preemptive scheduling. Existing analysis techniques for non-preemptive scheduling in uniprocessor systems [22, 8, 10, 16], use complex response time analyses with exponential running time complexities. An extension to holistic analysis to account for resource blocking due to non-preemptive scheduling in distributed systems is presented in [9]. This adjustment for non-preemptive scheduling makes this analysis more pessimistic than holistic analysis applied to preemptive scheduling. The paper also compares this technique with delay bounds obtained using network calculus [5, 6], and concludes that the worst case response time predicted by holistic analysis, tends to be superior to that of network calculus. The holistic analysis technique, however, assumes knowledge of all task routes and execution times in the system, which could be difficult or expensive to obtain in a large distributed setting. In contrast to such techniques, the meta-schedulability test presented in this paper allows the use of any uniprocessor schedulability analysis technique under preemptive scheduling, to be used to analyze the distributed system scheduled using non-preemptive scheduling. Thus, the complexity of the proposed analysis is directly dependent on the complexity of the single stage analysis technique used. Further, to estimate the worst case end-to-end delay of a task, the meta-schedulability test only uses task execution time information along the route of the task under consideration. Notwithstanding the reduced computational complexity and the lesser dependence on task execution information, our simulation studies in Section 7 show that the meta-schedulability test applied to non-preemptive scheduling performs significantly better than preemptive scheduling analyzed using holistic analysis. We believe that this paper can open the door to more widespread study and use of non-preemptive scheduling in distributed systems.

The remainder of this paper is organized as follows. Section 2 briefly describes the system model, states the main result, and outlines some intuitions into the delay composition theorem. In Section 3, this theorem is proved. Section 4 constructs a transformation of the DAG under non-preemptive scheduling into an equivalent hypothetical single stage system under preemptive scheduling. In Section 5, the proposed delay composition theorem and transformation to a single stage are extended to allow arbitrary fixed priority scheduling at different stages of the DAG. Using the aforementioned transformation to a uniprocessor system, in Section 6, we illustrate how to use single stage schedulability analyses to analyze DAGs. In Section 7, we present results of simulation experiments that show the extent to which non-preemptive scheduling can perform better than preemptive scheduling in terms of admissible utilization, while still meeting all deadlines of tasks. We also show conditions under which non-preemptive scheduling performs better than preemptive scheduling, and conditions under which the opposite is true. Related work is reviewed in Section 8. We conclude in Section 9 with directions for future work.

2. System Model and Problem Statement

The system model we consider is a multi-stage distributed data processing system. Periodic or aperiodic tasks arrive at this system and require execution on a set of resources (such as processors[1]), each performing one stage of task execution. Specifically, we consider a multi-stage distributed data processing system, whose topology can be expressed as a Directed Acyclic Graph (DAG). An edge in the DAG between stage i and stage j, indicates that a task that completes execution on stage i, could move on to execute at stage j. For the sake of deriving a general delay composition theorem that applies to periodic and aperiodic tasks alike, we do not make any implicit periodicity assumptions and consider individual task invocations in isolation. We call these invocations, *jobs*. We assume that the priority of each job is the same across all the stages at which it executes. We relax this assumption later in Section 5. In a given system, many different jobs may have the same priority (e.g., invocations of the same task in fixed-priority scheduling). However, there is typically a tie-breaking rule among such jobs (e.g., FIFO). Taking the tie-breaker into account, we can assume without loss of generality that each individual job has its own priority. This assumption will simplify the notations used in the derivations.

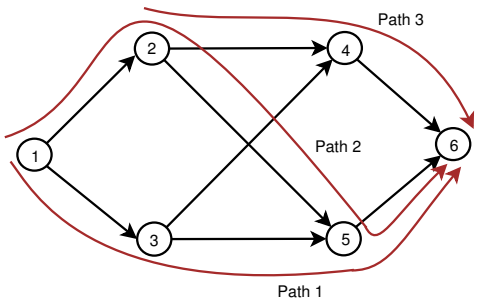

Figure 1. Example directed acyclic graph with job-flow paths, and a feasible number assignment for stages

A job can enter the system at any stage, request processing on a sequence of stages (a path in the DAG), and leave the system at any stage. Figure 1 shows an example of a directed acyclic distributed processing system along with a few sample job-flow paths. Let the total number of stages be $N^{'}$. We number these stages from 1 to $N^{'}$, in the order visited by the jobs. In other words, if there exists an edge in the DAG between stages i and j, then $i < j$. Such a numbering always exists as there are no cycles in the topology. However, this numbering may not be unique, and we choose any one number assignment that satisfies the above mentioned condition. Let $Path_i$ denote the set of stages comprising the path chosen by job J_i in the system. Let $A_{i,j}$ be the arrival time of job

[1] While we equate a resource to a processor, the same discussion applies to other resources such as network links and disks as long as they are scheduled in priority order. A distributed system can thus contain heterogeneous resources that include processing, communication and disk I/O stages.

J_i at stage j, where $1 \leq j \leq N^{'}$. The arrival time of the job to the entire system, called A_i, is the same as its arrival time to its first stage, $A_i = A_{i,t}$, where t is the smallest numbered stage in $Path_i$. Let D_i be the end-to-end (relative) deadline of J_i. It denotes the maximum allowable latency for J_i to complete its computation in the system. Hence, J_i must exit the system by time $A_i + D_i$. The computation time of J_i at stage j, referred to as the *stage execution time*, is denoted by $C_{i,j}$, for $1 \leq j \leq N^{'}$. If a job J_i does not execute at stage j, that is $j \notin Path_i$, then $C_{i,j}$ is zero. Let $S_{i,j}$, called the *stage start time*, be the time at which J_i starts executing on a stage j, and let $F_{i,j}$, called the *stage finish time*, be the time at which J_i completes executing on stage j.

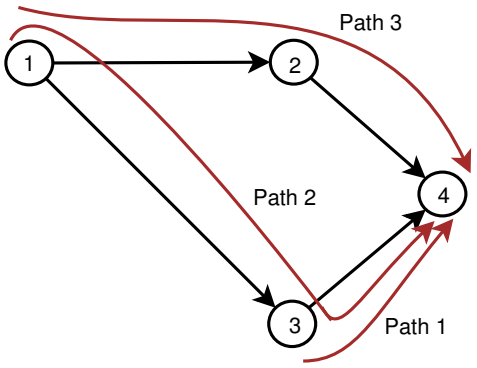

Figure 2. A sub-DAG chosen from the larger DAG, assuming J_1 follows path 3; stages are re-numbered

The purpose of this paper is to advocate the use of non-preemptive scheduling in distributed real-time systems. We provide an intuition as to why non-preemptive scheduling can achieve superior system utilization compared to preemptive scheduling, under certain conditions where the computation times of different jobs are not too dissimilar, while still meeting all the deadlines of jobs. The main theoretical contribution of this paper lies in deriving a delay composition theorem for non-preemptive scheduling that bounds the worst case delay experienced by any job as a function of the execution times of other jobs that have a common execution stage with it. This delay composition theorem for non-preemptive scheduling leads to a superior analysis technique that can admit a greater system utilization compared to any known admission controller for preemptive scheduling systems. We shall now state the delay composition theorem, and then provide some crucial insights.

Let the job whose delay is to be estimated be J_1, without loss of generality. As we are interested in the delay of J_1, we need to only consider those stages that may potentially influence the delay of J_1. We consider a stage i, only if stage i is reachable in the DAG from the first stage at which J_1 executes, and the last stage at which J_1 executes is reachable from stage i. We remove all stages that do not satisfy this condition, and renumber the stages from 1 through N ($N \leq N'$). By the above definition, stage 1 is the first stage at which J_1 executes, and stage N is the last stage at which J_1 executes. As before, it is ensured that if there exists an edge between stage i and stage j, then $i < j$. Note that considering only a subset of the stages constructs a sub-graph of the original DAG, and is therefore still a DAG. The job-flow path of each job J_i is accordingly truncated, to only consider the sub-path belonging to the chosen sub-DAG. Figure 2 shows such a sub-DAG, assuming J_1 follows path 3 of the DAG shown in Figure 1. Let S denote the set of all jobs that have execution intervals in the system between J_1's arrival and finish time, and have some common execution stage with J_1 (S includes J_1). Let $\bar{S} \subseteq S$ denote the set of all jobs with higher priority than J_1 and including J_1, and let $\underline{S} \subset S$ denote the set of all jobs with lower priority than J_1. Jobs that do not have a common execution stage with J_1 do not affect the delay of J_1. Let $C_{i,max}$, for any job J_i, denote its largest stage execution time, on stages where both J_i and J_1 execute.

We define a *split-merge* between the paths of jobs J_i and J_1, as a scenario where the path of J_i splits from the path of J_1, and intersects (merges with) the path of J_1 at a later stage. In more concrete terms, if there exists consecutive stages $j_1, j_2, \ldots, j_k (k \geq 2)$ in the path of J_1, and of these stages only j_1 and j_k belong to the path of J_i, and there is at least one other stage j' ($j_1 < j' < j_k$) on which J_i executes, then a split-merge is said to exist between J_i and J_1. Figure 2 shows a split-merge between paths 2 and 3. The total number of split-merges between the paths of J_i and J_1 is denoted by $SM_{i,1}$. Let $M_k(j) \subseteq \underline{S}$ denote the set of jobs with a lower priority than job J_k, whose paths merge with the path of J_k at stage j.

The delay composition theorem for J_1 is stated as follows:

Non-preemptive DAG Delay Composition Theorem. *Assuming a non-preemptive scheduling policy with the same priorities across all stages for each job, the end-to-end delay of a job J_1 in an N-stage DAG can be composed from the execution parameters of other jobs that delay it (denoted by set S) as follows:*

$$Delay(J_1) \leq \sum_{i \in \bar{S}} C_{i,max}(1 + SM_{i,1}) + \sum_{\substack{j \in Path_1 \\ j \leq N-1}} \max_{i \in S}(C_{i,j}) + \sum_{j \in Path_1} \max_{i \in M_1(j)} C_{i,max} \quad (1)$$

Observe that, from the perspective of deriving the delay composition theorem, we are not concerned (for the moment) with how to determine set S (or set $\bar{S}$). We are merely concerned with proving the fundamental property of delay composition over any such set. From the perspective of schedulability analysis, however, it is useful to estimate a worst case S and $\bar{S}$ to compute worst-case delay. Trivially, in the worst case, S would include all jobs J_i whose active intervals $[A_i, A_i + D_i]$ overlap that of J_1 (i.e., overlap $[A_1, A_1 + D_1]$). This is true because a job J_i whose deadline precedes the arrival of J_1 or whose arrival is after the deadline of J_1 has no

execution time intervals between J_1's arrival time and deadline (in a schedulable system), and hence cannot be part of S. The use of the delay composition theorem for schedulability analysis is further elaborated in Section 4.

Some comments are warranted on the form of the delay composition theorem. Observe that the first term is a summation over all higher priority jobs, and is proportional to the amount of 'traffic' that merges with the job under consideration. As this term is proportional to the number of higher priority jobs in the system and is independent of the number of stages in the system, we call this the *job-additive* component of J_1's delay. The second and third terms are a summation over the stages on which J_1 executes, and is independent of the number of jobs in the system. For this reason, we call this the *stage-additive* component of J_1's delay. In other words, if jobs and stages were thought of as two orthogonal dimensions, the summation is carried out only across one dimension at a time. The result is therefore much lower than a summation in both dimensions at the same time (i.e., adding up *all* single stage execution times of all jobs on all stages). The latter summation would reflect the worst-case delay of J_1 if overlap between stage executions was not taken into account.

As a first observation, it is interesting to note that preemption can reduce execution overlap among stages. For example, consider the case of a two-job system, where both jobs execute on all stages, shown in Figure 3. In Figure 3(i), the higher-priority job J_i arrives together with J_1 and is given the (first-stage) CPU. When J_i moves on to the second stage, J_1 can execute in parallel on the first. However, as shown in Figure 3(ii), if J_i arrives *after* J_1 and preempts it, when J_i moves on to the next stage, only the *unfinished* part of J_1 on the stage where it was preempted can overlap with J_i's execution on the next stage. In other words, execution overlap is reduced and J_1 takes longer to finish than it did in the previous case. The key difference between the two cases is the preemption of the lower priority job by the higher priority job, which caused the execution overlap between successive stages in the distributed execution to reduce. A question that naturally follows from this observation is whether non-preemptive scheduling can perform better than preemptive scheduling for distributed systems. Figure 3(iii) shows the execution of the two tasks for the same arrival times as in Figure 3(ii), but when non-preemptive scheduling is used. Notice that job J_1 finishes much earlier under non-preemptive scheduling, and J_i is only marginally delayed. Thus, the overall system throughput is improved. This observation that non-preemptive scheduling can perform better than preemptive scheduling for distributed systems, is true only when the execution times of jobs are relatively similar. Figure 3(iv), for example illustrates a scenario where the higher priority job J_i is blocked for a significantly long duration, waiting for the lower priority job J_1 to complete execution. This is clearly undesired behavior.

The above example uncovers a very fundamental scheduling problem in distributed systems. There are certain situations where non-preemptive scheduling yields a better worst case performance than preemptive scheduling, and there are situations where the opposite is true. Although in this paper we do not mathematically quantify the conditions under which one is better than the other, we take a first step towards understanding why and to what extent non-preemptive scheduling can have better worst case performance than preemptive scheduling. In Section 7, we study and characterize through simulations, the space in which non-preemptive scheduling outperforms preemptive scheduling in distributed directed acyclic systems.

It is important to note that the observation that preemption increases end-to-end delay and hinders schedulability, is purely an artifact of distributed execution where a job executes on multiple stages in a sequential manner, and does not apply to uniprocessor systems. It is well known that for uniprocessor systems, preemptive scheduling is better than non-preemptive scheduling in terms of schedulability of jobs. This stigma against non-preemptive scheduling in uniprocessor systems, has perhaps been carried forth onto distributed systems too. This paper aims to break this misconception, and hopes to stimulate research and use of non-preemptive scheduling in distributed systems. With the intuitions provided above, we proceed to prove the delay composition theorem.

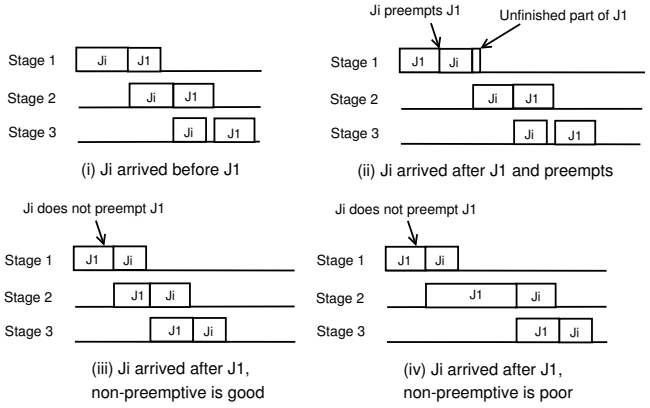

Figure 3. Figure showing the possible cases of two jobs in the system.

3. Delay Composition for Non-Preemptive Scheduling in Directed Acyclic Systems

A higher priority job J_i is said to *overtake* a lower priority job J_1 at a stage j, if it either did not require execution or executed after J_1 on stage $j-1$ (or the first stage prior to stage j on which J_1 executed), but executed before J_1 on stage j. Before we proceed to prove the delay composition theorem, we first prove a simple helper lemma.

Lemma 1. *The number of times a higher priority job J_i can overtake J_1 is at most one more than the number of split-merges between the paths of J_i and J_1 ($SM_{i,1}$).*

Proof. The proof is by a simple induction on the number of split-merges between the paths of J_i and J_1. The basis step is when there are no split-merges, $SM_{i,1} = 0$. In this case, J_i can overtake J_1 at most once, as after J_i overtakes J_1, it will always execute ahead of J_1 on every future stage, as the priorities are the same on all stages. Once the paths of J_i and J_1 split, the path of J_i never intersects the path of J_1.

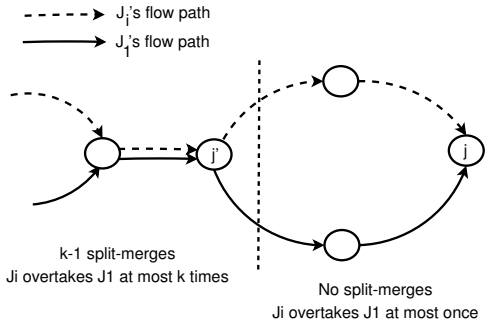

Figure 4. Figure illustrating proof of Lemma 1.

Assume that the lemma is true for all $SM_{i,1} \leq k-1$, for some $k \geq 1$. To prove the result for $SM_{i,1} = k$. Let stage j be the last stage where both J_i and J_1 execute. As $SM_{i,1} \geq 1$, there exists a stage $j' < j$, where the paths of J_i and J_1 split. Further, let stage j' be the last such split in the paths of J_i and J_1. Figure 4 illustrates this scenario. Up to and including stage j', the number of split-merges is $k-1$, and hence from induction assumption, the number of times J_i overtakes J_1 up to stage j' is at most k. Starting from stage $j'+1$, there are no split-merges in the paths of J_i and J_1 (the last split occurs at stage j'). From the basis step, the number of times J_i overtakes J_1 beyond stage $j'+1$ is at most one. Therefore, when $SM_{i,1} = k$, J_i overtakes J_1 at most $k+1$ times. ∎

The delay composition theorem is proved in two phases. First, we consider only higher priority jobs. In the presence of only higher priority jobs, the delay composition theorem can be proved by induction on task priority. We first prove the theorem for a two-job scenario (Lemma 2). We then prove the induction step, where we assume that the delay composition theorem is true for $k-1$ jobs, $k \geq 3$, add a k^{th} job with highest priority (with arbitrary job-flow path), and prove that the delay composition theorem still holds. We then account for lower priority jobs, and show that regardless of the number of lower priority jobs, the increase in delay due to lower priority jobs as a result of resource blocking is only proportional to the number of stages in the distributed system, and not proportional to the number of lower priority jobs.

Lemma 2. *When J_1 and J_2 are the only two jobs in the system, and J_2 has a higher priority than J_1, the delay experienced by J_1 is at most*

$$Q = \sum_{i=1}^{2} C_{i,max}(1 + SM_{i,1}) + \sum_{\substack{j \in Path_1, \\ j \leq N-1}} \max_{i=1,2}(C_{i,j}) \quad (2)$$

Proof. From Lemma 1, the maximum number of times J_2 can overtake and delay J_1 is $1 + SM_{2,1}$. In order to obtain a worst case delay for J_1, we assume that J_2 overtakes J_1 every time the paths of the two jobs meet (the case where J_2 does not overtake J_1 after some split-merge can be easily shown to cause a lower delay for J_1).

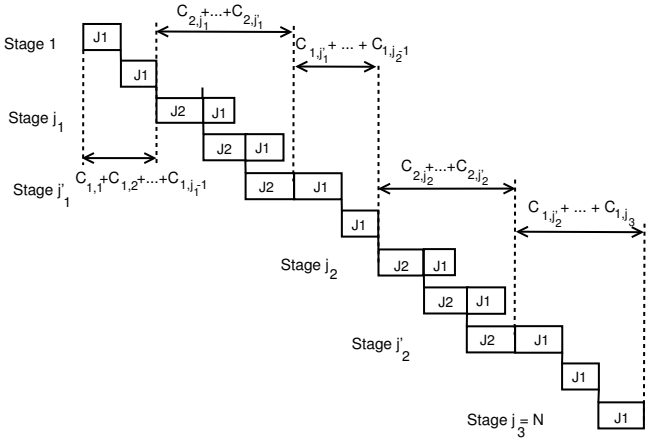

Figure 5. Figure showing the delay for job J_1, illustrating Lemma 2.

Let the stages at which J_2 overtakes J_1 be $j_1, j_2, \ldots, j_{SM_{2,1}+1}$. Each such 'overtake' occurs after a split-merge in the paths of J_2 and J_1. Let stage j'_k be a stage between j_k and j_{k+1} in the path of job J_1 ($j_{k+1} > j'_k \geq j_k$), such that j'_k is the last stage before stage j_{k+1} where J_1 waits for J_2, for $1 \leq k < SM_{2,1}+1$. For $k = SM_{2,1}+1$, j'_k is the last stage where J_1 waits for J_2 before completing its execution in the system. For notational simplicity, define stage $j_{SM_{2,1}+2}$ to be stage N. By definition of j'_k, J_1 does not wait for J_2 between stages j'_k and j_{k+1}, for $1 \leq k \leq SM_{2,1}+1$. Figure 5 illustrates the delay experienced by J_1. Until stage j_1, job J_1 does not wait for job J_2. The delay of J_1 up to the time J_1 is overtaken on stage j_1 is at most $C_{1,1} + \ldots + C_{1,j_1-1}$ (in the worst case, J_2 overtakes J_1 when J_1 has almost completed execution on the stage prior to j_1). Starting from the time J_1 is overtaken on stage j_k and until J_1 starts execution on stage j'_k, for all k, the delay is given by

$$\sum_{\substack{t \in Path_1 \\ j_k \leq t \leq j'_k}} C_{2,t}$$

Stage j'_k is the last stage where J_1 waits for J_2 before another split-merge occurs and J_2 overtakes J_1. Starting from J_1's execution on stage j'_k and up to the time J_1 is overtaken on stage j_{k+1} (or completes execution in the system), the delay is given by

$$\sum_{\substack{t \in Path_1 \\ j'_k \le t < j_{k+1}}} C_{1,t}$$

Notice that, in the above expressions for the delay of J_1, there is at most one execution time of a job on every stage 1 through N (in the path of J_1), except for stages j'_k, $1 \le k \le SM_{2,1}+1$, which contain two execution times, one from each job. To compute a delay bound, let us replace one per-stage computation time at each of the stages up to $N-1$ (that belong to $Path_1$) by $\max_{i=1,2} C_{i,t}$ for that stage. The delay of J_1 can therefore be written as,

$$\begin{aligned} Q &\le \sum_{k=1}^{SM_{2,1}+1} C_{2,j'_k} + \Big(\sum_{\substack{t \in Path_1 \\ t \le N-1}} \max_{i=1,2} C_{i,t}\Big) + C_{1,N} \\ &\le C_{2,max}(1+SM_{2,1}) + C_{1,max} + \sum_{\substack{t \in Path_1 \\ t \le N-1}} \max_{i=1,2} C_{i,t} \quad (3) \end{aligned}$$

Inequality 3 follows from the fact that stages j'_k (for every k) contribute an execution time of job J_2, each of which is less than $C_{2,max}$, and there are $(SM_{2,1}+1)$ such terms. Stage N contributes an execution time of $C_{1,N} \le C_{1,max}$, and $SM_{1,1} = 0$. This proves the lemma. ∎

We shall now prove the delay composition theorem for DAGs in the presence of higher priority jobs only, by induction on job priority.

Lemma 3. *Assuming a non-preemptive scheduling policy with the same priorities across all stages for each job, the end-to-end delay of a job J_1 of lowest priority in a distributed DAG with $n-1$ higher priority jobs is at most*

$$Delay(J_1) \le \sum_{i=1}^{n} C_{i,max}(1+SM_{i,1}) + \sum_{\substack{t \in Path_1 \\ t \le N-1}} \max_{i=1}^{n}(C_{i,t})$$

Proof. Without loss of generality, we assume that a job J_i has a higher priority than a job J_k, if $i > k, i, k \le n$. That is, J_n has the highest priority, and J_1 has the least priority.

The basis step is the case when there are only two jobs in the system, J_1 and J_2. The delay composition theorem for two jobs is precisely Lemma 2.

Assume that the result is true for $n = k-1$ jobs, $k \ge 3$. That is,

$$Delay_{k-1}(J_1) \le \sum_{i=1}^{k-1} C_{i,max}(1+SM_{i,1}) + \sum_{\substack{t \in Path_1 \\ t \le N-1}} \max_{i \le k-1} C_{i,t} \quad (4)$$

We need to show the result when a k^{th} job J_k, with highest priority and arbitrary flow path, is added. Let L_k be a system with k jobs, with arbitrary arrival times for each of the jobs. Let L_{k-1} be the system without job J_k, and with the same arrival times and flow paths for all the other jobs as in system L_k.

The number of split-merges in the paths of J_k and J_1 is $SM_{k,1}$. By breaking the path of J_1 after each split in the paths of J_k and J_1, the path of J_1 can be split into $SM_{k,1}+1$ parts. In each of these parts, J_k can overtake J_1 at most once. A key observation here is that job J_k in these parts, can be thought of as $SM_{k,1}+1$ independent jobs $J_{k_1}, J_{k_2}, \ldots, J_{k_{SM_{k,1}+1}}$. Each J_{k_i} executes in the i^{th} part ($1 \le i \le SM_{k,1}+1$), and does not meet J_1 at any of the other parts. We shall show that for every J_{k_i}, the job-additive component of J_1's delay due to J_{k_i} is at most $C_{k,max}$. The delay composition theorem when job J_k is added will follow naturally.

We now consider three cases. The first case considers that J_{k_i} arrived before (or together with) J_1 to the first common stage where J_{k_i} completed execution before J_1, and such a first common stage is stage 1 (this can happen only for $i=1$). The second case generalizes the first case, so that the first common stage where J_{k_i} completed execution before J_1 can be any stage $j > 1$. The case where J_{k_i} arrives later and overtakes J_1, is considered as the third case.

Case 1: Stage 1 is the first common stage between J_{k_i} and J_1, and J_{k_i} does not overtake J_1.

If J_{k_i} executed after J_1 on some stage and never overtakes J_1, it does not cause any delay to J_1. Therefore, it is safe to assume that J_{k_i} arrived before or together with J_1 to stage 1 and to every subsequent common stage. Note that, if there exists an idle time between the execution of J_{k_i} and J_1 on some stage j, the delay of J_1 on stage j is independent of the execution time of J_{k_i} (and other jobs that execute before the idle time) on stage j. Therefore, beyond the last stage j, where there is no idle time between the execution of J_{k_i} and J_1, J_{k_i} will not influence the delay of J_1. After J_{k_i} completes execution on stage j, the delay of J_1 in system L_k is identical to its delay in the system L_{k-1}, with only $k-1$ tasks and starting from stage j. Therefore, the delay of J_1 can be expressed as the delay up to the time J_{k_i} completes execution on stage j (J_k arrives before J_1 to the system), added to the worst case delay of J_1 in system L_{k-1} starting from stage j (as shown in Equation 5). This is shown in Figure 6.

$$\begin{aligned} Delay_k(J_1) &= F_{1,N} - A_{1,1} \\ &= (F_{1,N} - F_{k_i,j}) + (F_{k_i,j} - A_{1,1}) \quad (5) \end{aligned}$$

As J_{k_i} arrived before J_1 to the system, the duration between the arrival of J_1 to the system ($A_{1,1}$) and the completion of J_{k_i}'s execution on stage j ($F_{k,j}$), is at most the time J_{k_i} takes to complete execution up to stage j ($F_{k_i,j} - A_{k_i,1}$) as shown in Inequality 6 (although, this induces pessimism, the pessimism is due to the lack of knowledge as to how much earlier J_{k_i} arrives compared to J_1; future work can attempt to quantify this difference in arrival times more accurately, to obtain a better bound). J_{k_i} is the highest priority job in the system, and does not wait to execute on any of the stages. The time for J_{k_i} to complete execution up to stage j is $(\sum_{t \in Path_1, t<j} C_{k_i,t}) + C_{k_i,j}$.

In addition to this, from induction assumption, the delay of J_1 from stages j through N is $\sum_{i=1}^{k-1} C_{i,max}(1 + SM_{i,1}) + \sum_{t \in Path_1, j \leq t \leq N-1} \max_{i \leq k-1}(C_{i,t})$ (Inequality 7). It should be noted that the delay composition theorem accounts for the delay of J_1 due to any worst case arrival pattern of higher priority jobs, and therefore applies to the arrival pattern of jobs at stage j in system L_k. Moreover, we are only concerned with the worst case delay of J_1, and we are not concerned at the moment about whether jobs meet their designated deadlines. In computing such a worst case delay, some higher priority jobs may be delayed even beyond their deadlines so as to inflict a worst case delay to J_1. Such a worst case arrival pattern may cause the system to be unschedulable, but the delay composition theorem does not concern itself with schedulability. Isolating the delay composition theorem from the notion of deadlines and schedulability, enables us to obtain an upper bound on the delay experienced by a job, purely in terms of computation times of higher priority jobs.

Thus,

$$Delay(J_1) \leq (F_{1,N} - F_{k_i,j}) + (F_{k_i,j} - A_{1,1})$$

$$\leq (F_{1,N} - F_{k_i,j}) + (F_{k_i,j} - A_{k_i,1}),\ as\ A_{k_i,1} \leq A_{1,1} \quad (6)$$

$$\leq \Big(\sum_{t \in Path_{k_i}, t<j} C_{k_i,t}\Big) + C_{k_i,j} + \sum_{i=1}^{k-1} C_{i,max}(1 + SM_{i,1}) + \sum_{\substack{t \in Path_1 \\ j \leq t \leq N-1}} \max_{i \leq k-1}(C_{i,t}) \quad (7)$$

$$\leq C_{k_i,j} + \sum_{i=1}^{k-1} C_{i,max}(1 + SM_{i,1}) + \sum_{\substack{t \in Path_1 \\ t \leq N-1}} \max_{i \leq k}(C_{i,t}) \quad (8)$$

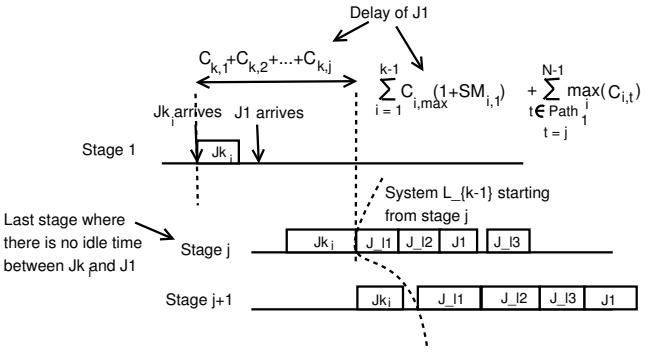

Figure 6. Figure showing the delay of J_1 for the case when J_k arrived before J_1 to the first common stage, which is stage 1.

Therefore, J_{k_i} delays J_1 by at most one maximum stage execution time (this is the job-additive component), where the maximum is over all stages where both J_{k_i} and J_1 execute ($j \in Path_{k_i} \cap Path_1$), apart from contributing to the maximum job execution times on stages 1 through j which belong to $Path_{k_i}$ (the stage-additive component).

Case 2: J_{k_i} arrived before or together with J_1 to the first common stage $j > 1$ where J_{k_i} completes execution before J_1.

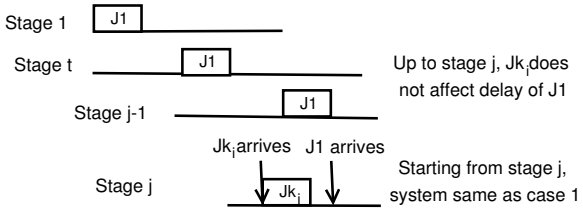

Figure 7. Figure showing the case when J_{k_i} arrived before J_1 to the first common stage j where J_{k_i} completed execution before J_1.

Up to stage j, the delay of J_1 is independent of job J_{k_i}, as J_{k_i} does not execute on these stages or executes after J_1. Starting from stage j, the system is identical to case 1, wherein the system can be thought of as one with $N - j + 1$ stages. Stage j is now the first stage in the system, and is also the first common stage where J_{k_i} and J_1 execute, and J_{k_i} arrived before or together with J_1 (shown in Fig 7). Therefore from case 1, J_{k_i} delays J_1 by at most one maximum stage execution time (the job-additive component), where the maximum is over all stages where both J_{k_i} and J_1 execute. Apart from the job-additive component, J_{k_i} contributes to the maximum job execution times on each stage (the stage-additive component) starting from stage j.

Case 3: J_{k_i} arrives after J_1 to the first common stage j where J_{k_i} completes execution before J_1, that is, J_{k_i} overtakes J_1.

As J_{k_i} is a part of J_k which does not have any split-merges with the path of J_1, J_{k_i} overtakes J_1 at most once. Until the time J_{k_i} overtakes J_1, the delay of J_1 is independent of J_{k_i}. Let J_{k_i} overtake J_1 at stage j. This implies that J_{k_i} arrived after J_1 to stage j and the two jobs were simultaneously in queue waiting for one or more other jobs to complete execution. Observe that allowing J_{k_i} to arrive just prior to J_1 at stage j, causes no difference to the interval of execution of J_1 on stage j, and hence causes no difference to the delay of J_1. Starting from stage j, J_{k_i} executes at each common stage before J_1. Therefore, the system starting from stage j can be thought of as one having $N - j + 1$ stages, and J_{k_i} arriving before J_1. We can then apply the result from case 2. Thus, J_1 is delayed by at most one maximum stage execution time of J_{k_i} (the job-additive component), apart from J_{k_i}'s contribution to the stage-additive component $\max_i(C_{i,t})$, for $j + 1 \leq t \leq N - 1$ (from Inequality 7). Figure 8 shows this scenario.

From the above three cases, each J_{k_i} adds to the delay of J_1 at most one maximum stage execution time of J_k. There are $1 + SM_{k,1}$ such jobs. Therefore, the total job-additive delay that J_k causes J_1 is at most $C_{k,max}(1 + SM_{k,1})$. Each J_{k_i} is part of the stage-additive component of J_1's delay. This delay is simply the sum of one maximum execution time over all jobs on each stage. The maximum of the execution

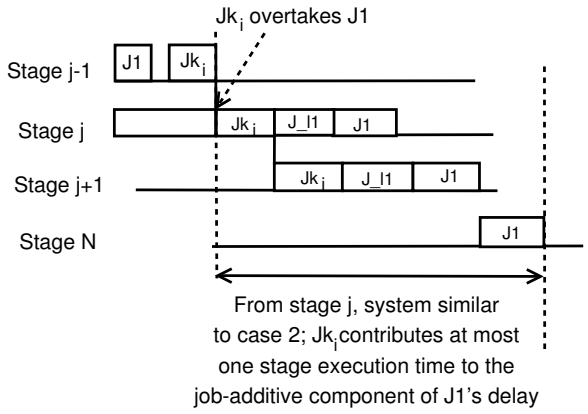

Figure 8. Figure showing the case when J_{k_i} arrived after J_1 and overtakes J_1 at stage j.

times of all J_{k_i} on each stage, is simply the execution time of job J_k on that stage. This is the contribution of J_k to the stage-additive component of J_1's delay, as in the expression of the delay composition theorem. This proves the induction step. Using this together with Lemma 2, Lemma 3 is proved. ■

Non-preemptive DAG Delay Composition Theorem. *Assuming a non-preemptive scheduling policy with the same priorities across all stages for each job, the end-to-end delay of a job J_1 in an N-stage DAG can be composed from the execution parameters of other jobs that delay it (denoted by set S) as follows:*

$$Delay(J_1) \leq \sum_{i \in \bar{S}} C_{i,max}(1 + SM_{i,1}) + \sum_{\substack{j \in Path_1 \\ j \leq N-1}} \max_{i \in S}(C_{i,j}) + \sum_{j \in Path_1} \max_{i \in M_1(j)} C_{i,max} \quad (9)$$

Proof. Lemma 3 proved the delay composition theorem in the presence of higher priority jobs alone. We shall now prove the theorem in the presence of both higher and lower priority jobs. Note that under preemptive scheduling lower priority jobs cause no delay to higher priority jobs. However, under non-preemptive scheduling, a higher priority job may block on a resource while a lower priority job is accessing it. In the worst case, a higher priority job may be delayed by at most one lower priority job at every stage in the distributed system. We characterize this delay using two cases - lower priority jobs whose paths merge with the path of J_1 at some stage, and lower priority jobs that execute together with, but ahead of the higher priority job J_1 on successive stages of the DAG (as in a pipeline).

Case 1: A lower priority job J_i whose path merges with the path of J_1 at some stage j, may arrive ahead of J_1 and block it at stage j. In the worst case, the lower priority job J_i would arrive at stage j, just before J_1 arrives at the stage, causing J_1 to wait for one complete stage execution time of J_i. Figure 9(i) illustrates such a scenario, where lower priority jobs J_i and J_k arrive just before J_1 to stages 2 and 3,

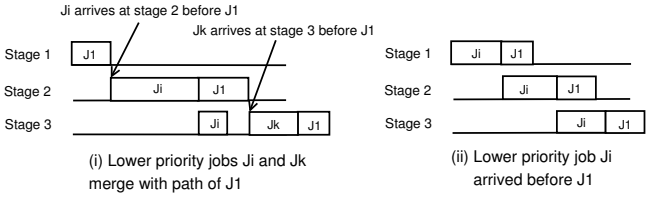

Figure 9. Figure illustrating the two different ways in which a lower priority job can delay J_1.

respectively, and cause J_1 to block. At each stage j in its execution path, job J_1 may block on at most one lower priority job J_i, whose path merges with the path of J_1 at that stage (that is, $J_i \in M_1(j)$). Therefore, in the worst case, J_1 is delayed by $\sum_{j=1}^{N} \max_{i \in M_1(j)} C_{i,max}$ (the reason for adding $C_{i,max}$ instead of just $C_{i,j}$ for each j, will be clear after the discussion of the next case).

Case 2: A lower priority job J_i that arrives ahead of J_1, and hence blocks J_1 at a stage j may continue to block J_1 at future stages (when there are no other jobs executing), as it will always arrive ahead of J_1 at each successive stage. Figure 9(ii) illustrates this scenario. Note that beyond the last stage j' where J_i executes ahead of J_1 and there is no idle time between the executions of J_i and J_1, the execution times of J_i have no impact on the delay of J_1. Between stages j and j', the execution of J_i on a stage takes place in parallel with J_1 on the previous stage. Therefore, similar to the proof of Lemma 2 and Lemma 3, it can be shown that J_i contributes to the stage additive component of J_1's delay, on each of the stages between j and j'. The delay due to J_i merging with J_1 at stage j (the delay from case 1) manifests itself only at stage j' and not at stage j (similar to the proofs of Lemma 2 and Lemma 3). As j' is not known, we upper bound this delay by adding $C_{i,max}$ for each such lower priority job.

From the above two cases, the delay of J_1 due to lower priority jobs alone is given by:

$$\sum_{\substack{j \in Path_1 \\ j \leq N-1}} \max_{i \in S}(C_{i,j}) + \sum_{j=1}^{N} \max_{i \in M_1(j)} C_{i,max}$$

In the proofs of Lemma 2 and Lemma 3, we assumed a worst case arrival pattern of higher priority jobs that cause a worst case delay to job J_1. This worst case arrival pattern of each higher priority job is independent of other jobs in the system, and is therefore applicable in the presence of lower priority jobs too. Similar to the proofs of Lemma 2 and Lemma 3, each higher priority job J_i may overtake J_1 at most $(1 + SM_{i,1})$ times, and each causes a worst case job-additive delay of $C_{i,max}$. Each higher priority job J_i also contributes towards the stage-additive component of J_1's delay, on the common execution stages between J_i and J_1. A detailed proof is omitted in the interest of brevity. This completes the proof of the delay composition theorem. ■

4. Reduction to a Single Stage System and Schedulability

In this section, we elucidate a systematic reduction of the schedulability problem in an acyclic distributed system under non-preemptive scheduling, to an equivalent uniprocessor preemptive scheduling problem using the delay composition theorem. Since delay predicted by the delay composition theorem grows with set S, let us first define the *worst-case* (i.e., largest) set S, denoted S_{wc}, of jobs that delay J_1. In this paper, we suggest a very simple (and somewhat conservative) definition of set S_{wc}. We expect that future work can improve upon this definition using more in-depth analysis. In the absence of further information, set S_{wc} is defined as follows.

Definition: The worst-case set S_{wc} of jobs that can potentially delay job J_1 (hence, include execution intervals between the arrival and finish time of J_1) includes all jobs J_i which have at least one common execution stage with J_1, and whose intervals $[A_i, A_i + D_i]$ overlap the interval where J_1 was present in the system, $[A_1, A_1 + delay(J_1)]$.

Observe that the above definition simply excludes the impossible, and is therefore a conservative definition. A job that does not have a common execution stage with J_1 can never delay J_1. Further, in a schedulable system, a job J_i that does not satisfy the above condition either completes prior to the the arrival of J_1 or arrives after its completion. Hence, it cannot possibly have execution intervals that delay J_1.

$$delay(J_1) \leq \sum_{i \in \bar{S}_{wc}} C_{i,max}(1+SM_{i,1}) + \sum_{\substack{j \in Path_1 \\ j \leq N-1}} \max_{i \in S_{wc}} C_{i,j} + \sum_{j \in Path_1} \max_{i \in M_1(j)} C_{i,max} \quad (10)$$

We construct an equivalent single stage system under preemptive scheduling by (i) replacing each job J_i in $\bar{S}_{wc}$ by an equivalent single stage job of execution time equal to $C_{i,max}(1 + SM_{i,1})$, and (ii) adding a lowest-priority job, J_e^* of execution time equal to $\sum_{j \in Path_1, j \leq N-1} \max_i(C_{i,j}) + \sum_{j \in Path_1} \max_{i \in M_1(j)} C_{i,max}$ (which are the last two terms in Inequality (10)), and deadline same as that of J_1. Note that the execution time of J_e^* includes the delay due to all lower priority tasks. Further, in the above reduction the hypothetical single stage system constructed is scheduled using preemptive scheduling, while the original DAG was scheduled using non-preemptive scheduling. This is because the higher priority jobs can overtake J_1 in the DAG, which corresponds to the equivalent higher priority jobs preempting J_e^* in the uniprocessor system. By the delay composition theorem, the total delay incurred by J_1 in the acyclic distributed system under non-preemptive scheduling is no larger than the delay of J_e^* on the uniprocessor under preemptive scheduling, since the latter adds up to the delay bound expressed on the right hand of Inequality (10).

For the case of periodic tasks, the delay bound can be significantly improved based on the observation that not all invocations of a higher priority task T_i can overtake an invocation of T_1, $1 + SM_{i,1}$ times. Let us suppose that during the execution of an invocation of T_1, at most x invocations of T_i overtake T_1 (x invocations are part of $\bar{S}_{wc}$). If one such invocation overtakes T_1 $1 + SM_{i,1}$ times, it implies that T_1 has progressed past the last split-merge between the paths of T_i and T_1, and therefore, future invocations of T_i can overtake T_1 at most once. Extending this argument, at most one invocation of T_i can overtake T_1 at each split-merge between the paths of T_i and T_1. Therefore, the maximum number of times x invocations of T_i can overtake T_1 is $x+SM_{i,1}$, rather than $x(1+SM_{i,1})$. Notice that the factor $SM_{i,1}$ now appears only once for each task, rather than once for each invocation of every task.

The reduction to a single stage system for periodic tasks can then be conducted by (i) replacing each periodic task T_i by an equivalent single stage task of execution time equal to $C_{i,max}$, and (ii) adding a lowest priority task with computation time equal to $\sum_{j \in Path_1, j \leq N-1} \max_i(C_{i,j}) + \sum_{j \in Path_1} \max_{i \in M_1(j)} C_{i,max} + \sum_{i \in \bar{S}_{wc}} C_{i,max} SM_{i,1}$.

For example, let us illustrate this transformation in the case of rate-monotonic scheduling of periodic tasks with periods equal to deadlines. Consider a set of periodic tasks, where each task T_i has a period P_i. As shown in Figure 10, there can be at most one invocation of each higher-priority task T_i in S_{wc} that arrives before an invocation of T_1, which causes a delay of at most $C_{i,max}$. The number of invocations of each task T_i that arrive after the invocation of T_1 and delay it, is no larger than $\lceil \frac{delay_1}{P_i} \rceil$. Following the reduction outlined above, then aggregating jobs of the same period into single periodic tasks, the following periodic task set is reached:

- Task T_e^* (of lowest priority), with a computation time $C_e^* = \sum_{i \in \bar{S}_{wc}} (C_{i,max} + C_{i,max} SM_{i,1}) + \sum_{j \in Path_1, j \leq N-1} \max_i(C_{i,j}) + \sum_{j \in Path_1} \max_{i \in M_1(j)} C_{i,max}$. In the above expression, in favor of simplicity, we abuse notation and use $\bar{S}_{wc}$ to denote the set of higher priority tasks, instead of higher priority invocations of tasks as defined earlier. The task T_e^* further has the same period and deadline as T_1 in the original set.
- Tasks T_i^*, each has the same period and deadline as one T_i in the original set, and has an execution time equal to $C_i^* = C_{i,max}$.

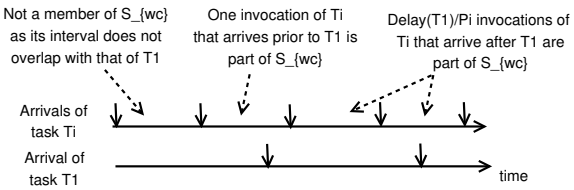

Figure 10. Invocations in S_{wc}.

Hence, if task T_e^* is schedulable using preemptive scheduling on a uniprocessor, so is T_1 on the original acyclic

distributed system under non-preemptive scheduling. The transformation is complete. In Section 6, we present DAG schedulability expressions for deadline monotonic scheduling based on the above task set reduction.

5. Extension to Any Scheduling Policy on Any Stage

The delay composition theorem assumed that each job is assigned the same priority on every stage of the distributed system. In this section, we relax this assumption. To analyze the delay of a job J_1 in a system L with an arbitrary scheduling policy at each stage, we transform the system into another system L' where each job has the same priority across all stages. We show that the delay of J_1 in the transformed system L' is no smaller than the delay of J_1 in the original system L. The delay composition theorem can then be applied to L' to obtain a bound on the delay of J_1 in L. The transformed system L' can also be reduced to a single stage system as outlined in Section 4, to analyze schedulability. Such a reduction would however assume that each stage applies an arbitrary fixed priority scheduling policy, where the priorities of different jobs (invocations) of a task are the same for a given stage, although a job can have different priorities on different stages.

All jobs in L are also jobs in L'. Only the priorities of the jobs at different stages is changed in L', and other parameters such as computation time and deadline are kept the same. The priorities of jobs at different stages is modified as follows: (i) Any job which has a higher priority than job J_1 at *any one* common stage of execution with J_1 in L, is assigned a higher priority than J_1 on all stages in L'; (ii) Any job which has a lower priority than J_1 on *all* common stages of execution with job J_1 in L, is assigned a lower priority than J_1 on all stages in L'. The relative priorities of higher priority jobs (and likewise of lower priority jobs) in L' can be assigned arbitrarily.

The delay of J_1 in such a transformed system L' can be bounded using the delay composition theorem. We shall now show that this also bounds the delay of J_1 in the original system L. We start from L', and using two simple transformations that do not increase the delay of J_1, we construct L. Thus, the delay of J_1 in L can be proved to be no larger than its worst case delay in L'.

For each stage in L', the jobs can be ordered based on their priorities. Consider any two jobs on a stage that are adjacent to each other in this ordering and have a higher priority than J_1. Notice that swapping the priorities of these two jobs does not affect the worst case delay of J_1. This is because the worst case delay of J_1 is only dependent on whether the two jobs have a higher priority than J_1, and not on the actual priorities of the two jobs. Further, the delay composition theorem is applicable to any arrival pattern of higher priority jobs, and therefore a change in the arrival pattern due to swapping the priorities, does not affect the worst case delay bound as per the delay composition theorem. Likewise, by successively swapping the priorities of jobs adjacent to each other, it is possible to construct the priorities of all jobs on all stages to resemble that in L, except for the priority of J_1. Let us call this system L'', where the relative priorities of all jobs on all stages are the same as in L, except J_1. J_1's priority on all stages is lower than any job which has a higher priority than it on at least one stage in L. By definition of L' (and also L''), J_1's priority on any stage in L' is no higher than its priority on the corresponding stage in L. Hence, the priority of J_1 in L'' can be increased at each stage to obtain system L. Increasing the priority of J_1 at each stage can only decrease its delay. Thus, starting from system L', by successively swapping the priorities of jobs we can obtain system L'' without increasing the delay of J_1. The priority of J_1 on different stages in L'' can be increased to obtain system L, which again does not increase the delay of J_1. This shows that the worst case delay of J_1 in system L' is no smaller than J_1's delay in system L. Using this transformation, the delay composition theorem can be applied to analyze schedulability under arbitrary fixed priority scheduling policies at each stage of the distributed system.

6. Utility of Derived Result

The reduction described in Section 4 enables large complex acyclic distributed systems under non-preemptive scheduling to be easily analyzed using any single stage schedulability analyses technique. In this respect, our solution is indeed a 'meta-schedulability test'. The only assumptions made by the reduction on the scheduling model are fixed priority preemptive scheduling, and tasks do not block for resources on any of the stages (i.e., independent tasks). In the rest of this section, we concern ourselves with schedulability analysis for periodic tasks. We assume that task T_k has a higher priority than task T_i, if $k < i$.

As examples, we show how the Liu and Layland bound [12] and the necessary and sufficient test based on response time analysis [1] can be applied to analyze periodic tasks in an acyclic distributed system. Other uniprocessor schedulability tests can be applied in a similar manner.

The Liu and Layland bound [12], applied to an acyclic distributed system under non-preemptive scheduling is:

$$\frac{C_e^*(i)}{D_i} + \sum_{k \leq i} \frac{C_k^*}{D_k} \leq i(2^{\frac{1}{i}} - 1)$$

for $1 \leq i \leq n$, where n is the number of tasks in the system, $C_k^* = C_{k,max}$; $C_e^*(i) = \sum_{k \leq i}(C_{k,max} + C_{k,max}SM_{k,i}) + \sum_{j \in Path_i, j \leq N-1} \max_{k \leq n}(C_{k,j}) + \sum_{j \in Path_i} \max_{k \in M_i(j)} C_{k,max}$. $C_{i,max}$ is the largest execution time of T_i on any stage, D_i is the end-to-end deadline, $M_i(j)$ is the set of tasks with lower priority than task i, whose path merges with task i at stage j.

The necessary and sufficient test for schedulability of periodic tasks under deadline monotonic scheduling proposed in [1], used together with our meta-schedulability test, will have the following recursive formula for the worst case response time R_i of task T_i:

$$R_i^{(0)} = C_e^*(i)$$
$$R_i^{(k)} = C_e^*(i) + \sum_{j<i} \left\lceil \frac{R_i^{(k-1)}}{P_j} \right\rceil C_j^*$$

The worst case response time for task T_i is given by the value of $R_i^{(k)}$, such that $R_i^{(k)} = R_i^{(k-1)}$. For the task set to be schedulable, for each task T_i, the worst case response time needs to be at most D_i.

7. Simulation Results

In this section, we evaluate our meta schedulability test using simulations. A custom-built simulator that models a distributed system with directed acyclic flows is used. As there are no previously known techniques to study aperiodic tasks under non-preemptive scheduling, we consider only periodic tasks in this evaluation. Each task requires processing at a fixed set of nodes in the distributed system. In order to maintain real-time guarantees within the system, an admission controller is used. The admission controller is based on a single stage schedulability test for deadline monotonic scheduling, such as the Liu and Layland bound [12] or response time analysis [1], together with our reduction of the multistage distributed system to a single stage, as shown in Section 6. Each periodic task that arrives at the system is tentatively added to the set of all tasks in the system. The admission controller then tests whether the new task set is schedulable. The new task is admitted if the task set is schedulable, and dropped if not.

Although the meta schedulability test derived in this paper is valid for any fixed priority scheduling algorithm, we only present results for deadline monotonic scheduling due to its widespread use. In the rest of this section, we use the term utilization to refer to the average per-stage utilization. Each point in the figures below represent average utilization values obtained from 100 executions of the simulator, with each execution running for 80000 task invocations. The default number of nodes in the distributed system is assumed to be 8. Each task on arrival requests processing on a sequence of nodes, with each node in the distributed system having a probability of RP (for Route Probability) of being selected as part of the route. The task's route is simply the sequence of selected nodes in increasing order of their node identifier. The default value of RP is chosen as 0.8. Note that all task routes are directed and acyclic. Deadlines (equal to the periods, unless explicitly specified otherwise) of tasks are chosen as $10^x a$ simulation seconds, where x is uniformly varying between 0 and DR (for deadline ratio), and $a = 500 * N$, where N is the number of stages in the task's route. Such a choice of deadlines enables the ratio of the longest task deadline to the shortest task deadline to be as large as 10^{DR}. If DR is chosen close to zero, tasks would have similar deadlines. If DR is higher (for example $DR = 3$), deadlines of tasks would differ more widely. The default value for DR is 0.5, and we refer to DR as the deadline ratio parameter. The execution time for each task on each stage was chosen based on the task resolution parameter, which is a measure of the ratio of the total computation time of a task over all stages to its deadline. The stage execution time of a task is calculated based on a uniform distribution with mean equal to $\frac{DT}{N}$, where D is the deadline of the task and T is the task resolution. The stage execution times of tasks were allowed to vary up to 10% on either side of the mean. Choosing the stage execution times to be nearly proportional to the end-to-end deadline, ensures that when tasks have similar deadlines (DR close to zero), then the execution times are also comparable. When tasks have widely different deadlines (a high value for DR), then the execution times are also widely varying. Our simulations validate our intuition presented in Section 2, that non-preemptive scheduling performs better than preemptive scheduling in the worst case when the task execution times are similar, and preemptive scheduling performs better than non-preemptive scheduling when the task execution times are different by more than a couple of orders of magnitude.

Under preemptive scheduling, task preemptions are assumed to be instantaneous, that is, the task switching time is zero. We used a task resolution of $1:100$. The default single stage schedulability test used is the response-time analysis technique presented in [1]. The 95% confidence interval for all the utilization values presented in this section is within 0.02 of the mean value, which is not plotted for the sake of legibility.

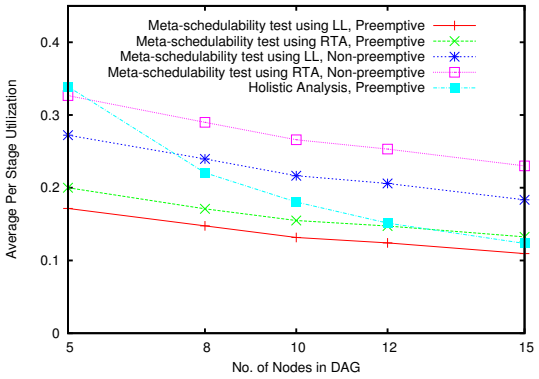

Figure 11. Comparison of meta-schedulability test using both preemptive and non-preemptive scheduling with holistic analysis for different number of nodes in the DAG

We first study the achievable utilization of our meta-schedulability test using both the Liu and Layland bound and response time analysis, for both preemptive as well as non-preemptive scheduling. We compare this with holistic analysis [20], applied to preemptive scheduling, for different number of nodes in the DAG, the results of which are shown in Figure 11. Even for an eight node DAG, non-preemptive scheduling analyzed using our meta-schedulability test significantly outperforms preemptive scheduling analyzed using both holistic analysis and our meta-schedulability test. A

major drawback of holistic analysis is that it analyzes each stage separately assuming the response times of tasks on the previous stage to be the jitter for the next stage. It therefore assumes that every higher priority job will delay the lower priority job at every stage of its execution, ignoring possible pipelining between the executions of the higher and lower priority jobs. This causes holistic analysis to become increasingly pessimistic with system size. As motivated in Section 2, preemption can reduce the overlap in the execution of jobs on different stages, resulting in non-preemptive scheduling performing better than preemptive scheduling in the worst case. It should also be noted that to estimate the delay and schedulability of a task, our meta-schedulability test only requires knowledge of task executions along the route of the task under consideration. In contrast, holistic analysis requires knowledge of all tasks in the entire system to analyze the schedulability of each task. Such global knowledge may be difficult or costly to obtain for most systems, especially with the growing scale and complexity of such systems.

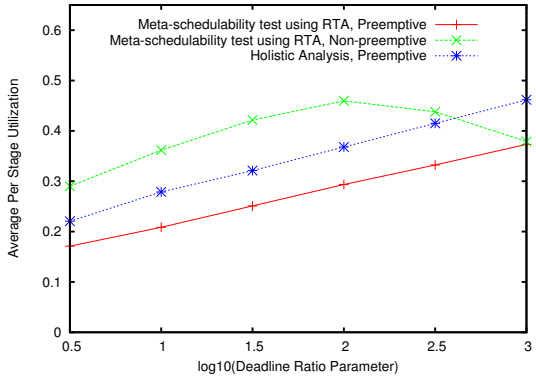

Figure 12. Comparison of meta-schedulability test using both preemptive and non-preemptive scheduling with holistic analysis for different deadline ratio parameters

To precisely evaluate the scenarios under which non-preemptive scheduling can perform better than preemptive scheduling in distributed systems, we conducted experiments varying the deadline ratio parameter while keeping the other parameters equal to their default values. A deadline ratio parameter DR value of x indicates that the end-to-end deadlines of tasks can differ by as much as 10^x. Stage execution times of tasks are chosen proportional to the end-to-end deadline. This implies that when the end-to-end deadlines of tasks are widely different, the lower priority tasks (those with large deadlines) have a large stage execution time. Initially, as DR increases, the utilization for both preemptive as well as non-preemptive scheduling increases, as lower priority tasks can execute in the background of higher priority tasks resulting in better system utilization. However, when DR increases, it implies that higher priority tasks can now be blocked for a longer duration under non-preemptive scheduling, which could lead to missed deadlines. Figure 12 plots a comparison of achievable utilization using the meta-schedulability test under both preemptive as well as non-preemptive scheduling with the achievable utilization using holistic analysis for different DR values ranging between 0.5 and 3.0. It can be observed from the figure that for small values of DR, non-preemptive scheduling results in better performance than preemptive scheduling. However, for values of DR greater than 2, that is, the end-to-end deadlines vary by over two orders of magnitude, preemptive scheduling performs better than non-preemptive scheduling. The increased blocking of higher priority tasks, causes the achievable utilization under non-preemptive scheduling to decrease beyond a DR value of 2.

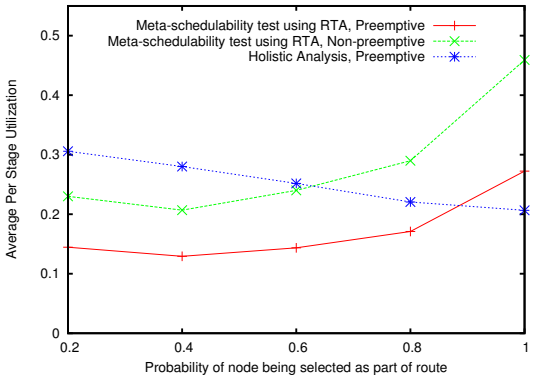

Figure 13. Comparison of meta-schedulability test using both preemptive and non-preemptive scheduling with holistic analysis for different route probabilities

We conducted a similar comparison of the three admission controllers as in the previous experiment, but for different values of the Route Probability (RP) parameter, which is the probability with which each node in the system is chosen as part of the route of each task. The RP parameter was varied from 0.2 to 1.0 in steps of 0.2. Note that the RP parameter of 1.0 denotes a perfectly pipelined system, where each task executes sequentially on all the nodes in the distributed system. For small values of RP, the number of stages on which each task executes is low. As observed in Figure 11, for lower number of execution stages, holistic analysis performs better than the meta-schedulability test. However, for larger values of RP, each task traverses more stages in the distributed system, causing holistic analysis to become more pessimistic in its worst case delay bound. The meta-schedulability test using non-preemptive scheduling performs the best for RP values greater than 0.6.

The above results have all been obtained by setting the end-to-end deadlines equal to the periods of tasks. Figure 14 plots a comparison of the meta-schedulability test under preemptive and non-preemptive scheduling with holistic analysis for different ratios of the end-to-end deadlines to the periods. When the ratio of the end-to-end deadline to period is higher, the laxity available to jobs is larger, and hence, the utilization of all the three analysis techniques are high. The meta-schedulability test under non-preemptive

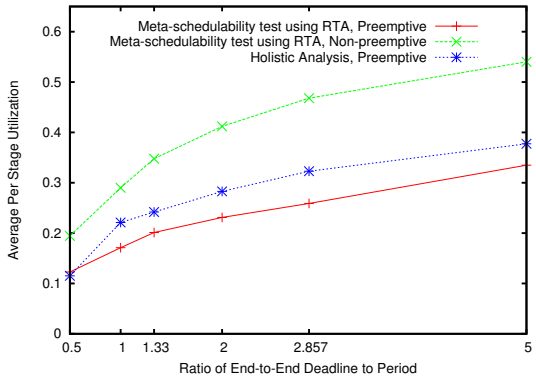

Figure 14. Comparison of meta-schedulability test using both preemptive and non-preemptive scheduling with holistic analysis for different ratios of end-to-end deadline to task periods

scheduling consistently outperforms preemptive scheduling analyzed using either the meta-schedulability test or holistic analysis.

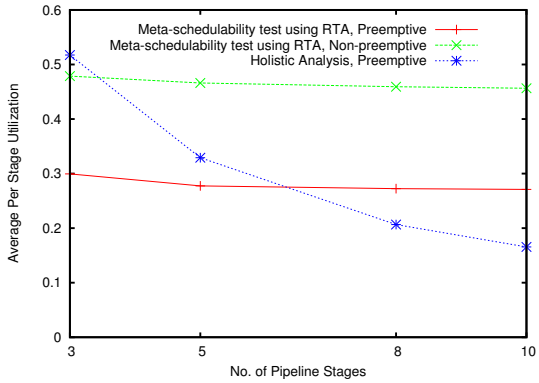

Figure 15. Comparison of meta-schedulability test using both preemptive and non-preemptive scheduling with holistic analysis for different number of pipeline stages

As pipelines are of special interest in several kinds of distributed systems such as web server farms, in the rest of this section, we present a comparison of non-preemptive and preemptive scheduling in pipelined systems, where every task is processed sequentially at every stage in the distributed system. We first evaluated the three admission controllers for different number of pipeline stages, the results of which are presented in Figure 15. The results are similar to those observed in the case of a DAG (Figure 11), except for the fact that the improvement in performance of the meta-schedulability test compared to holistic analysis is more significant. The reason for this is that flows can no longer merge at different points in a task's execution in the distributed system. The meta-schedulability test assumes that whenever the paths of two tasks merge, the higher priority task will inflict a worst case delay on the lower priority task. This pessimistic assumption is neutralized in a strictly pipelined system, enabling the meta-schedulability test to perform better. The meta-schedulability test under non-preemptive scheduling performs significantly better than the other two admission controllers.

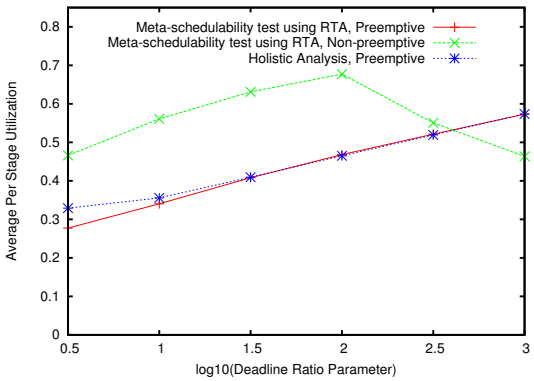

Figure 16. Comparison of meta-schedulability test using both preemptive and non-preemptive scheduling with holistic analysis for different deadline ratio parameters in a 5 stage pipeline

Similar to the study of the three admission controllers in DAGs varying the deadline ratio parameter DR, to precisely quantify the space in which non-preemptive scheduling performs better than preemptive scheduling, we compare the performance of the admission controllers in a pipeline by varying DR. Figure 16 plots this comparison. The trends observed here are very similar to those observed in Figure 12. The utilization of all three techniques increase initially with DR. For DR values greater than 2, the utilization under non-preemptive scheduling decreases, as higher priority jobs are now blocked for longer durations.

We also conducted experiments varying the task resolution parameter, which is the ratio of the average per-stage computation times of tasks to their end-to-end deadline. As the utilization admitted by the three techniques did not vary significantly with different task resolution parameters, the results of this study are not presented here.

8. Related Work

The first study of feasible regions in real-time systems, was first conducted by Liu and Layland in their seminal work [12], where under certain specific restrictions, they presented utilization bounds for uniprocessor systems. These utilization bounds were extended to multiprocessor systems in [3]. Resource constraints were considered and a single-stage utilization bound which was less pessimistic that the Liu and Layland bound was presented. While these utilization bounds were sufficient conditions for schedulability, exact tests such as [1, 11] were also proposed.

Several scheduling algorithms have been proposed for statically scheduling precedence constrained tasks in dis-

tributed systems [18, 21, 7]. Given a set of periodic tasks, such algorithms attempt to construct a schedule of length equal to the least common multiple of the task periods. The schedule will accurately specify the time intervals during which each task invocation will be executed. Needless to say, such algorithms have a huge time complexity and are clearly unsuitable for complex, large scale distributed systems, where simplicity is of essence.

Analyzing the Worst Case Execution Times (WCET) of tasks in processor and memory pipeline architectures is a well studied problem in the area of real-time operating systems ([23, 19] and references thereof). Such algorithms execute in time that is exponential in the number of tasks in the system. Further, the approach would be difficult to implement in a distributed setting and is more error-prone.

A few offline schedulability tests have also been proposed for distributed systems. These techniques divide the end-to-end deadline into individual per-stage deadlines, and tend to ignore the overlap that exists between the execution of different stages. A distributed pipeline framework was presented in [4]. Offset-based response time analysis techniques for EDF were proposed in [15, 17] which divide the end-to-end deadline into individual stage deadlines. Recently, [24] designed and implemented a middleware layer based on deferrable servers for aperiodic tasks with hard end-to-end deadlines in distributed real-time applications. Techniques to divide the end-to-end deadline into sub-deadlines for individual stages were presented. A technique that combines offline and online scheduling is proposed in [14]. Here, precedence and communication constraints are converted offline into per-stage pseudo deadlines for each task. Online scheduling is then used to efficiently determine feasibility.

Holistic schedulability analysis for distributed hard real-time systems was first proposed in [20]. Here, the worst case delay at a stage is taken as the jitter for the next stage. While this technique does not divide the end-to-end delay into sub-deadlines for individual stages, it nevertheless analyzes each stage separately, and does not account for the overlap in the execution of tasks at different stages in the distributed system.

In stark contrast to preemptive scheduling, non-preemptive scheduling has received very little attention from the real-time community. A major reason for this is due to the fact that non-preemptive scheduling performs worse than preemptive scheduling for uniprocessor systems. In this paper, we have shown that this conclusion is not always true for distributed systems.

Complex response time analyses with exponential running time complexities are used in [22, 8, 10, 16] to analyze uniprocessor systems with non-preemptive scheduling. An extension to holistic analysis in distributed systems to account for blocking due to non-preemptive scheduling is presented in [9]. The paper presents a comparison of this analysis technique with network calculus [5, 6], and concludes that the worst case response time as predicted by the holistic analysis technique tends to be superior to that of network calculus in most cases. As this holistic analysis technique tends to be more pessimistic than holistic analysis for preemptive scheduling, we do not evaluate this technique in this paper. In contrast to such techniques, we reduce the problem of analyzing a multistage pipelined system with non-preemptive scheduling to that of analyzing a single stage system using preemptive scheduling. Thus, well known tests such as the Liu and Layland test and response time analysis can be adopted to analyze multistage systems that use non-preemptive scheduling, resulting in more efficient schedulability analysis.

In [13], non-preemptive scheduling has been shown to not be robust even for uniprocessor systems, that is, increasing the processor speed can cause a previously schedulable task set to become unschedulable. The analysis technique presented in this paper, is an offline analysis technique which bounds the *worst case delay* of tasks under non-preemptive scheduling in distributed systems. As the proposed bound is a worst case bound, if the task set is deemed to be schedulable using the meta-schedulability test, increasing the processor speed would not undermine schedulability (likewise, other existing schedulability techniques that determine a worst case bound, are not affected by this non-robustness property of non-preemptive scheduling).

9. Conclusions and Future Work

In this paper, we have shown that in distributed systems, non-preemptive scheduling can perform better than preemptive scheduling in terms of task schedulability, under certain circumstances. We consider a distributed system, where the job flow paths form a Directed Acyclic Graph (DAG). In such a system, we derive a worst-case delay bound for tasks scheduled using non-preemptive scheduling. The delay bound leads to a transformation of the DAG to an equivalent uniprocessor system scheduled using preemptive scheduling. This transformation enables the wealth of uniprocessor schedulability analysis to be used to analyze distributed systems scheduled using non-preemptive scheduling. Our simulation studies characterize the situations under which non-preemptive scheduling performs better than preemptive, and also the situations under which the opposite is true. We believe this new result can foster more extensive study and use of non-preemptive scheduling in distributed systems.

This work opens the door for schedulability theory research in distributed systems in multiple directions. For example, in [2], earliest-effective-deadline-first was shown to be an optimal non-preemptive scheduling policy for distributed systems, when the execution times of tasks are the same across all the stages of the distributed system. The delay composition rule can help in the search for an optimal (or near-optimal) scheduling policy for distributed systems with arbitrary task characteristics. The delay composition rule could aid the study of obtaining optimal rate control, routing and scheduling policies in distributed systems and large networks. The current work addresses only directed acyclic systems and does not account for loops. Accounting for loops

can enable the result to be applied to semi-conductor chip manufacturing plants, where chips revisit the same service center multiple times before exiting the system. Further, extending the result to multi-resource systems can help widen the applicability of the result.

References

[1] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, pages 284–292, 1993.

[2] R. Bettati and J. W. Liu. Algorithms for end-to-end scheduling to meet deadlines. In *IEEE Symposium on Parallel and Distributed Processing*, pages 62–67, December 1990.

[3] E. Bini, G. Buttazzo, and G. Buttazzo. A hyperbolic bound for the rate monotonic algorithm. In *Euromicro Conference on Real-Time Systems*, pages 59–66, June 2001.

[4] S. Chatterjee and J. Strosnider. Distributed pipeline scheduling: End-to-end analysis of heterogeneous multi-resource real-time sytems. In *IEEE International Conference on Distributed Computing Systems*, pages 204–211, May 1995.

[5] R. Cruz. A calculus for network delay, part i: Network elements in isolation. *IEEE Transactions on Information Theory*, 37(1):114–131, January 1991.

[6] R. Cruz. A calculus for network delay, part ii: Network analysis. *IEEE Transactions on Information Theory*, 37(1):132–141, January 1991.

[7] G. Fohler and K. Ramamritham. Static scheduling of pipelined periodic tasks in distributed real-time systems. In *9th Euromicro Workshop on Real-Time Systems*, pages 128–135, June 1997.

[8] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *IEEE Real-Time Systems Symposium*, pages 129–139, December 1991.

[9] A. Koubaa and Y.-Q. Song. Evaluation and improvement of response time bounds for real-time applications under non-preemptive fixed priority scheduling. *International Journal of Production and Research (IJPR)*, 42(14):2899–2913, July 2004.

[10] I. Kuroda and T. Nishitani. Asynchronous multirate system design for programmable dsps. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 5, pages 549–552, March 1992.

[11] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium*, pages 166–171, December 1989.

[12] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of ACM*, 20(1):46–61, 1973.

[13] A. K. Mok and W.-C. Poon. Non-preemptive robustness under reduced system load. In *IEEE Real-Time Systems Symposium*, pages 200–209, December 2005.

[14] M. D. Natale and J. A. Stankovic. Dynamic end-to-end guarantees in distributed real-time systems. In *Proc. Real-Time Systems Symposium*, pages 216–227, December 1994.

[15] J. Palencia and M. Harbour. Offset-based response time analysis of distributed systems scheduled under edf. In *Euromicro Conference on Real-Time Systems*, pages 3–12, July 2003.

[16] T. M. Parks and E. A. Lee. Non-preemptive real-time scheduling of dataflow systems. In *IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 5, pages 3235–3238, May 1995.

[17] R. Pellizzoni and G. Lipari. Improved schedulability analysis of real-time transactions with earliest deadline scheduling. In *IEEE Real Time and Embedded Technology and Applications Symposium (RTAS)*, pages 66–75, March 2005.

[18] K. Ramamritham. Allocation and scheduling of complex periodic tasks. In *IEEE International Conference on Distributed Computing Systems*, pages 108–115, May 1990.

[19] J. Schneider. Cache and pipeline sensitive fixed priority scheduling for preemptive real-time systems. In *IEEE Real-Time Systems Symposium*, pages 195–204, November 2000.

[20] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Elsevier Microprocessing and Microprogramming*, 40(2-3):117–134, 1994.

[21] J. Xu and D. Parnas. On satisfying timing constraints in hard real-time systems. *IEEE Transactions on Software Engineering*, 19(1):70–84, January 1993.

[22] X. Yuan and A. K. Agrawala. A decomposition approach to non-preemptive scheduling in hard real-time systems. In *IEEE Real-Time Systems Symposium*, pages 240–248, December 1989.

[23] N. Zhang, A. Burns, and M. Nicholson. Pipelined processors and worst case execution times. *Real-Time Systems*, 5(4):319–343, 1993.

[24] Y. Zhang, C. Lu, C. Gill, P. Lardieri, and G. Thaker. End-to-end scheduling strategies for aperiodic tasks in middleware. Technical Report WUCSE-2005-57, University of Washington at St. Louis, December 2005.