Transforming Distributed Acyclic Systems into Equivalent Uniprocessors Under Preemptive and Non-Preemptive Scheduling

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

Abstract

Many scientific disciplines provide composition primitives whereby overall properties of systems are composed from those of their components. Examples include rules for block diagram reduction in control theory and laws for computing equivalent circuit impedance in circuit theory. No general composition rules exist for real-time systems whereby a distributed system is transformed to an equivalent single stage analyzable using traditional uniprocessor schedulability analysis techniques. Towards such a theory, in this paper, we extend our previous result on pipeline delay composition to the general case of distributed acyclic systems as well as to non-preemptive scheduling. The new extended analysis provides a worst-case bound on the end-to-end delay of a job under both preemptive as well as non-preemptive scheduling, in a distributed system described by a Directed Acyclic Graph (DAG). The bound is computed as a function of graph topology and resource sharing policies on different resources. Our composition rule permits a simple transformation of the distributed task system into an equivalent uniprocessor task-set analyzable using traditional uniprocessor schedulability analysis. Hence, using the transformation described in this paper, the wealth of theory available for uniprocessor schedulability analysis can be easily applied to a larger class of distributed systems.

1. Introduction

With the increasing complexity and scale of real-time computing artifacts, efficient tools are needed for schedulability analysis in distributed real-time systems. Rigorous theory exists today for schedulability analysis of uniprocessors and multiprocessors, while mostly heuristics are used to analyze larger arbitrary-topology systems. This raises the question of whether a formal transformation can be found that converts a given distributed system into an equivalent uniprocessor system analyzable using the wealth of existing uniprocessor schedulability theory. Such transformations are not uncommon in other contexts. For example, control theory describes transformations that reduce complex block diagrams into an equivalent single block that can be analyzed for stability and performance properties. Similar rules (e.g., *Kirchoff Laws*) exist in circuit theory.

In an earlier milestone paper [10], the authors derived a delay composition rule that provided a bound on the end-to-end delay of jobs in a pipelined distributed system. A transformation of the pipelined system to an equivalent uniprocessor system based on the delay composition rule was then shown. Motivated by the ultimate goal of a general transformation theory, this paper extends the results in [10] in three main directions. First, we extend the results to a larger class of distributed systems; namely, those described by arbitrary Directed Acyclic Graphs (DAG). Second, we analyze non-preemptive scheduling (in addition to preemptive scheduling), and show that, in certain situations, non-preemptive scheduling can in fact result in better performance than preemptive scheduling in distributed systems. Third, we extend the results to accommodate resource partitioning (in lieu of priority-based scheduling). For example, TDMA is a very common resource allocation mode in real-time communication networks. TDMA is an example of partitioning a resource among different principals as opposed to allocating it in some priority order by a scheduling policy.

In our system model, each task traverses a path of multiple stages of execution and must exit the system within specified end-to-end latency bounds. The combination of all such paths forms a DAG. Application domains for such systems include manufacturing plants, data processing backends, and communication in sensor networks. For both preemptive and non-preemptive scheduling, we derive a *delay composition rule* that describes how delay composes across tasks and across stages in a distributed system. The rule is expressed as a worst-case end-to-end delay bound of a task invocation as a function of per-stage execution times of other task invocations along its path. This leads to a natural reduction of the multi-stage system to a hypothetical single-stage system scheduled using preemptive scheduling. A wide range of existing schedulability analysis techniques can now be applied to the new preemptively scheduled uniprocessor task set, to analyze the original distributed system under both preemptive and non-preemptive scheduling. We then show how the above mentioned results can be applied to tasks,

*The work reported in this paper was supported in part by the National Science Foundation under grants CNS 06-13665, CNS 06-15318, and CNS 05-53420

whose subtasks themselves form a directed acyclic graph (as opposed to a path as considered earlier).

Fundamentally, there are two ways resources can be shared among tasks. The first is by resource partitioning, where each task gets a dedicated share of the resource. Cyclic executives (that assign a fixed CPU time to each task) and TDMA channel allocation are examples of this policy in real-time systems. The second sharing method is by priority. In this case, the resource is allocated to tasks in priority order. Our theory accommodates both resource sharing methods. The only assumption on the scheduling policy made by our composition rule is that it assigns the same priority to a task invocation at all stages. Resources scheduled in priority order (e.g., processors) can be mixed with resources shared by partitioning (e.g., real-time communication media). No assumptions are made on the periodicity of the task set. Different invocations of the same task need not have the same priority. Hence, the delay composition rule and the corresponding system transformation (to an equivalent uniprocessor) apply to static-priority scheduling (such as rate-monotonic), dynamic-priority scheduling (such as EDF) and aperiodic task scheduling alike, as well as partitioned-resource systems.

The delay composition rule provides a bound on the worst case end-to-end delay of a task using only information of computation times of other tasks *along its route*. In contrast, traditional schedulability analysis, such as holistic analysis [17], require global knowledge of task routes and computation times, in order to predict the worst case end-to-end delay. With the growing size of distributed and embedded systems, the need for such global knowledge could make such analysis difficult or expensive to scale. We show that the schedulability analysis technique developed in this paper outperforms existing techniques by a factor that grows larger with system scale. For small distributed systems, existing literature is adequate.

We show that under certain conditions non-preemptive scheduling can perform better than preemptive scheduling in distributed systems. The explanation, as we show later, is because non-preemptive scheduling has a "smoothing" effect on task arrivals at downstream stages because task completion times on a processor are separated by entire computation times and not fractions thereof. This smoothing improves resource utilization. Preemption, in contrast, allows for more bursty arrival scenarios, hence, reducing schedulability in the worst case. Using simulations, we provide a preliminary characterization of the space in which non-preemptive scheduling performs better than preemptive scheduling, and also the situations wherein the opposite is true. We hope that this observation will foster more extensive 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 crucial intuitions into the preemptive and non-preemptive versions of the delay composition theorem. In Section 3, an outline of the proof of this theorem is provided. We show how partitioned resources can be handled in Section 4. In Section 5, we discuss the application of the delay composition theorem to schedulability analysis and show how well-known single stage schedulability analysis techniques can be used to analyze acyclic distributed systems. We describe, in Section 6, a flight control system as an example application, where the theory developed in this paper can be applied. In Section 7, we describe how the system model can be extended to include tasks whose subtasks themselves form a DAG. In Section 8, we compare the performance of schedulability analysis based on our delay composition theorem with holistic analysis, and characterize the conditions under which non-preemptive scheduling can result in higher system utilization than preemptive scheduling. Related work is reviewed in Section 9. We conclude in Section 10.

2. System Model and Problem Statement

In this paper, we consider a multi-stage distributed system that serves several classes of real-time tasks. The execution of each task follows a sequential path through the system. In other words, tasks require service at a sequence of resource stages, where each resource could be shared either by partitioning or by scheduling using some priority order. A very common example is one where tasks are scheduled in priority order on processors that share communication resources in a TDMA or token-passing fashion. Partitioning communication resources among senders using a TDMA or token-passing protocol is a common approach for ensuring temporal correctness in distributed real-time systems. It is therefore the primary manner in which network resources are addressed in this work. Of course certain other communication protocols such as the CAN bus do support priority scheduling directly and can also be analyzed (represented by a prioritized resource as opposed to a partitioned resource). Notably, randomized protocols such as the wireless Ethernet are outside the scope of the current framework. Such protocols, however, are usually not suitable for real-time applications.

The union of all paths followed by tasks in the system forms a Directed Acyclic Graph (DAG). This system model can be extended to include tasks whose subtasks themselves form a DAG, instead of a sequential path. We later extend this system model, in Section 7, to include tasks whose subtasks themselves form a DAG, instead of a sequential path. Tasks can be periodic or aperiodic. 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. In order to ensure that the delay composition theorem is general enough to apply to both periodic and aperiodic tasks, we make no implicit periodicity assumptions and consider individual task invocations (jobs) in isolation. We assume that the priority order of any two jobs is the same across all the stages that are scheduled in priority order (not partitioned), at which these jobs execute. Since priority-based policies sequence tasks in a manner that depends only on the order of priorities and not their exact values, without loss of generality, we assume that the priority of each job is identical on all stages and that job priorities are unique. Ties between

jobs that have the same priority (e.g., invocations of the same task in fixed-priority scheduling) can be broken using any tie-breaking rule (e.g., FIFO). This assumption will simplify the notations used in the derivations.

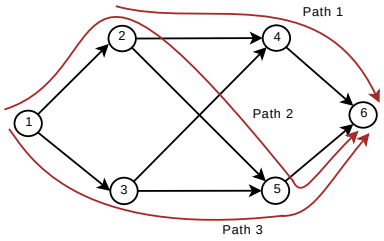

Figure 1. Example DAG with job-flow paths, and a feasible number assignment for stages

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 system along with a few sample job-flow paths. Let the total number of stages be $N^{'}$. We number these stages from 1 to $N^{'}$, such that 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. Let $Path_i$ denote the sequence of stages comprising the path chosen by job J_i in the system. Let the arrival time of job J_i to the system, that is, its arrival to its first stage, be called A_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. 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.

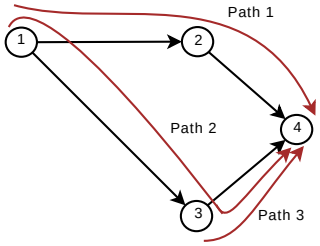

Figure 2. A sub-DAG of the larger DAG, assuming J_1 follows path 1; stages are re-numbered

In the rest of this section, we formally state the delay composition theorem for directed acyclic systems, under both preemptive as well as non-preemptive scheduling. For the purpose of stating and proving the delay composition theorem, we first assume that at each stage, access to the resource is scheduled in priority order. We relax this assumption later in Section 4 by showing that partitioned resources can be reduced to (slower) priority-based resources plus a delay. Hence, for the purposes of computing a worst-case delay bound, it is sufficient to consider prioritized resource scheduling only.

We further provide crucial insights into the delay composition theorem, and motivate 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.

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 1 of the DAG shown in Figure 1. Let S denote the set of all jobs with 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. 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 1. 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 under preemptive scheduling is stated as follows:

Preemptive DAG Delay Composition Theorem. *Assuming a 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 jobs that preempt or delay it (denoted by set $\bar{S}$) as follows:*

$$Delay(J_1) \leq \sum_{i \in \bar{S}} 2C_{i,max}(1+SM_{i,1}) + \sum_{\substack{j \in Path_1 \\ j \leq N-1}} \max_{i \in \bar{S}}(C_{i,j}) \quad (1)$$

The delay composition theorem for J_1 under non-preemptive scheduling 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:

$$Delay(J_1) \leq \sum_{i \in \bar{S}} C_{i,max}(1 + SM_{i,1}) + \sum_{\substack{j \in Path_1 \\ j \leq N-1}} \max_{i \in S}(C_{i,j}) + \sum_{j \in Path_1} \max_{i \in M_1(j)} C_{i,max} \quad (2)$$

Note that the two delay composition theorems are not concerned with defining set S (or $\bar{S}$), or in determining the schedulability of jobs. They simply provide a fundamental bound on the end-to-end delay of jobs, over any given set S. However, in order to conduct schedulability analysis using the delay composition rule, it is imperative to determine a worst case set S. As a trivial (and therefore pessimistic) baseline, S can include the set of all jobs that are present concurrently in the system with job J_1. More formally, the worst-case set S can be defined as the set of jobs J_i, whose active intervals $[A_i, A_i + D_i]$ overlap that of J_1 (i.e., overlap $[A_1, A_1 + D_1]$). Better sets can be constructed given more information about the task set (e.g., if one assumes periodic tasks). We further elaborate on schedulability analysis motivated by the delay composition theorem in Section 5.

Some comments are warranted on the form of the two delay composition theorems. Observe that the first term under both preemptive and non-preemptive scheduling, is a summation over all higher priority jobs. This term is proportional to the amount of higher priority traffic that 'merges' with the job under consideration (J_1), as one maximum stage execution time of each higher priority job J_i is accounted for every time J_i merges with J_1 ($SM_{i,1}$ denotes the number of times the path of J_i merges with the path of J_1). As this term is proportional to the number of higher priority jobs and is independent of the number of stages in the system, we call this the *job-additive* component of J_1's delay. Notice that the multiplicative factor "2" in the case of preemptive scheduling is not present under non-preemptive scheduling. The intuition behind this will be explained shortly. The second term under both preemptive and non-preemptive scheduling is a summation over the stages on which J_1 executes, and is independent of the number of jobs in the system. Note that the maximization is applied over all higher priority jobs ($\bar{S}$) for preemptive scheduling, and over all jobs (S) for non-preemptive scheduling. Even lower priority jobs are considered under non-preemptive scheduling, as a lower priority job may block a higher priority job from accessing a resource. Further, lower priority jobs that merge with the path of J_1, may cause J_1 to block on the resource if it arrived before J_1. J_1 would have to wait for at most one such lower priority job at each stage where lower priority jobs merge with J_1. This is accounted for in the third term of the non-preemptive delay composition theorem, and is not present in the preemptive version. The second and third terms in the non-preemptive case, and the second term in the preemptive case, is called the *stage-additive* component of J_1's delay.

In [10], we showed that preemption can reduce the execution overlap among stages (this is the reason for the factor 2 under preemptive scheduling, which is not present under non-preemptive scheduling). We revisit the motivating example provided in [10], and extend it to intuitively explain why non-preemptive scheduling can sometimes perform better than preemptive scheduling. Consider the case of a two-job system, where both jobs execute on all stages, as shown in Figure 3. Let J_i arrive before or together with J_1 and start executing on the first stage (shown in Figure 3(i)). Notice that when J_i moves on to execute on the second stage, J_1 can execute in parallel on the first. However, 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 (as shown in Figure 3(ii)). This reduces the execution overlap (i.e., amount of concurrent execution) between stages. Consequently, the lower-priority job 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 be reduced. 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. Depending on their respective deadline values, schedulability could actually be 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 evinces a very interesting observation. There are certain situations where non-preemptive scheduling yields 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 step towards understanding why and to what extent non-preemptive scheduling can have better worst-case performance than preemptive scheduling. In Section 8, we study and characterize through simulations, the space in which non-preemptive scheduling outperforms preemptive scheduling in distributed directed acyclic systems.

With the intuitions provided above, we proceed to sketch the proofs of the delay composition theorems.

3. Delay Composition for Directed Acyclic Systems

In Sections 3.1 and 3.2, we sketch the proofs for the preemptive and non-preemptive versions of the delay composi-

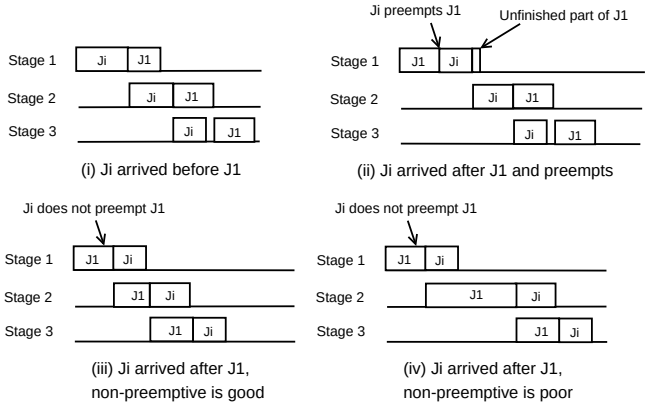

Figure 3. Figure showing the possible cases of two jobs in the system.

tion rule, respectively. In Section 3.3, we show how the delay composition theorem reduces to the special case of pipelines presented in [10].

A higher priority job J_i is said to *overtake* a lower priority job J_1 at a stage j, if it arrived after J_1, but executed before it 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 preempts (or overtakes) 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. Informally, the lemma is correct because for J_i to preempt J_1 more than once, it needs to execute on a stage where J_1 does not execute and then arrive (merge) at some stage after J_1 arrives at that stage. Once J_i preempts J_1 and in the absence of any further split-merge, J_i will always arrive ahead of J_1 on the remaining stages and cannot cause further preemptions.

The formal 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 preempt (or overtake[1]) J_1 at most once, as after J_i preempts 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, and J_i will cause no further preemptions.

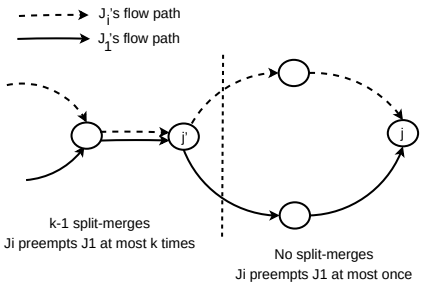

Figure 4. Figure illustrating proof of Lemma 1.

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 preempts 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 preemptions beyond stage $j'+1$ is at most one. Thus, when $SM_{i,1} = k$, J_i preempts J_1 at most $k+1$ times. □

In this paper, we only sketch the outline of the proofs of the delay composition theorems (elaborate proofs can be found in our technical reports [9, 8]. The proofs are by induction on task priority, similar to the proof of the pipeline delay composition theorem [10], and we only highlight the additional parts of the proof. Without loss of generality, we assume that a job J_i has a higher priority than a job J_k, if $i > k, i, k \leq n$. That is, J_n has the highest priority, and J_1 has the lowest priority.

3.1 Proof Outline for the Preemptive Case

Preemptive DAG Delay Composition Theorem. *Assuming a 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*

$$Delay(J_1) \leq \sum_{i=1}^{n} 2C_{i,max}(1 + SM_{i,1}) + \sum_{\substack{t \in Path_1 \\ t \leq N-1}} \max_{i=1}^{n}(C_{i,t})$$

Proof. Consider a higher priority job J_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 preempt 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 ($1 \leq i \leq SM_{k,1}+1$), and does not meet J_1 at any of the other parts. For every J_{k_i}, the job-additive component of J_1's delay due to J_{k_i} is at most $2C_{k,max}$ (we do not elaborate on the proof of this statement, as this is very similar to the proof of the pipeline delay composition theorem [10]). There are $(SM_{k,1}+1)$ such parts. The stage-additive component is the maximum execution times over all higher priority jobs accrued over all stages. The delay composition theorem under preemptive scheduling then follows naturally. □

3.2 Proof Outline for the Non-Preemptive Case

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, similar to the preemptive case (as outlined in Lemma 2). 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

[1] In the rest of this proof, the word 'preempt' can be replaced by 'overtake', for the case of non-preemptive scheduling

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. *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*

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

Proof. The proof of this lemma is very similar to the proof of the preemptive case (Section 3.1) and the proof of the pipeline delay composition theorem. However, there is one main difference which needs to be carried forth throughout the proof. As motivated in Section 2, the multiplicative factor 2 in the first term of the delay composition theorem is not present in the non-preemptive case. Each time a higher priority job J_i overtakes J_1, the job-additive component of J_1's delay is at most *one* maximum stage execution time of J_i, and is independent of the number of stages. From Lemma 1, the number of times J_i overtakes J_1 is at most $(1 + SM_{k,1})$. Therefore, the total job-additive component of J_1's delay is at most $C_{i,max}(1 + SM_{i,1})$ for each higher priority job J_i. The stage-additive component is the sum of one maximum stage execution time of any task accrued over all stages. □

Apart from the multiplicative factor 2, that does not appear in the non-preemptive case, the other major difference compared to the preemptive case is the presence of lower priority jobs that can block and delay J_1. We now proceed to characterize the delay due to lower priority jobs and prove the theorem in its entirety.

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:*

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

Proof. Lemma 2 proved the delay composition theorem in the presence of higher priority jobs alone, and accounted for the first two terms of the delay composition theorem. 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).

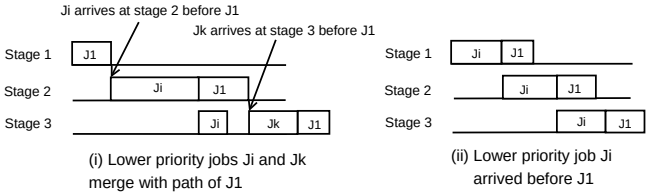

Figure 5. Illustration of two different ways in which a lower priority job can delay J_1.

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 5(i) illustrates such a scenario, where lower priority jobs J_i and J_k arrive just before J_1 to stages 2 and 3, 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 5(ii) illustrates this scenario. Similar to the proof of the pipeline delay composition theorem [10], it can be shown that J_i contributes to the stage additive component of J_1's delay, on each of the stages on which J_i and J_1 execute. The delay due to J_i merging with J_1 at stage j (the delay from case 1) manifests itself only at the last stage where J_1 waits for J_i to complete execution and not at stage j (as in the proof of the pipeline delay composition theorem). 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, 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. A detailed proof is omitted in the interest of brevity. This completes the proof sketch of the delay composition theorem for the non-preemptive case. □

3.3 Special Case of Pipelined Distributed Systems

We now show how the pipeline delay composition theorem [10] is a special case of the preemptive DAG delay composition theorem derived in this paper. In a strictly pipelined

system, every job traverses through the same sequence of stages, and hence the number of split-merges between the paths of any two jobs is zero, i.e., $SM_{i,1} = 0$ for all jobs J_i. Therefore, the preemptive DAG delay composition theorem (Inequality 1) reduces to,

$$Delay(J_1) \leq \sum_{i=1}^{n} 2C_{i,max} + \sum_{t \leq N-1} \max_{i=1}^{n}(C_{i,t})$$

Notice that, the right hand side of this bound is at least as large as the pipeline bound, as $2C_{i,max} \geq C_{i,max1} + C_{i,max2}$, where $C_{i,max1}$ and $C_{i,max2}$ are the largest and second largest stage execution times of job J_i, respectively. Thus, the pipeline delay composition theorem derived in [10] is a special case of the preemptive DAG delay composition theorem.

4. Handling Partitioned Resources

The delay composition theorem as described so far, is only applicable to systems where resources are scheduled in priority order. However, resources such as network bandwidth are often *partitioned* amongst the contestants using a reservation policy or TDMA protocol. In such a partitioned resource, a contestant may access the resource only during its reserved time-slot or token. Reservations or partitions could also be created for classes of tasks. Resource sharing within a class is achieved using prioritized scheduling. Similarly, a node, in its reserved communication time-slot, will send messages on the wire in priority order. We shall therefore assume this hierarchical (priority-scheduling within partition) model for the partitioned resources. In this section, we show how a partitioned resource can be translated to a corresponding resource scheduled in priority order, so that the delay of any task in the latter case is no smaller than its delay in the former. Once this transformation is conducted for all partitioned resources in the distributed system, the delay composition theorem can be used to determine the worst case end-to-end delay.

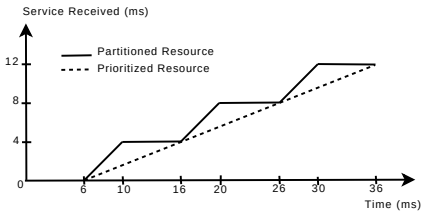

Figure 6. Illustration of conversion of a partitioned resource into a prioritized resource.

Consider a stage j with a partitioned resource. Let a task T_i belong to a class c, that is allotted $B_{c,j}$ time units of the resource every $B_{total,j}$ time units. Task T_i requires a total of $C_{i,j}$ time units to complete execution at stage j. In the worst case, it would arrive at the stage just when its slot is over. It would then have to wait for $B_{total,j} - B_{c,j}$ time units before its slot. After this initial delay, tasks of class c obtain $B_{c,j}$ units of the resource every $B_{total,j}$ time units. For the purpose of analyzing the delay of tasks belonging to class c, we can model the partitioned resource as a prioritized resource consisting of only class c tasks that provides service at a rate slower by a factor $\frac{B_{c,j}}{B_{total,j}}$ compared to the original service rate, and causing an additional delay of $B_{total,j} - B_{c,j}$ time units. Figure 6 illustrates the service received by a class of tasks over time for the original partitioned resource and the its corresponding hypothetical prioritized resource, when $B_{c,j} = 4ms$ and $B_{total,j} = 10ms$. Note that the service received under the prioritized resource will always be lesser than in the partitioned resource, causing tasks to be delayed longer. When analyzing the end-to-end delay of T_i, the computation time of T_i in the new hypothetical prioritized resource can be taken as $C_{i,j} \times \frac{B_{total,j}}{B_{c,j}} + (B_{total,j} - B_{c,j})$ (the additional delay is subsumed in the computation time). The computation time of all other tasks T_k of class c would be $C_{k,j} \times \frac{B_{total,j}}{B_{c,j}}$.

Once this transformation is conducted for all partitioned resources that T_i encounters in the system, the delay composition theorem can be directly applied to compute the worst case end-to-end delay of T_i.

5. Transforming Distributed Systems

Using the pipeline delay composition theorem, in [10] we showed how the analysis of a multi-stage pipeline can be transformed into that of an equivalent single stage system. A similar reduction of the DAG to an equivalent single stage system can be conducted using the preemptive and non-preemptive DAG delay composition theorems too. The worst case set S, denoted S_{wc}, of jobs that delay or preempt J_1, is defined similar to the definition in [10]:

Definition: The worst-case set S_{wc} of all jobs that delay or preempt 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, $[A_1, A_1 + delay(J_1)]$.

In Sections 5.1 and 5.2, we show how a different equivalent uniprocessor system can be created to analyze the schedulability of each task in the original system. This is not unlike the traditional way rate monotonic analysis deals with tasks with resource blocking, where effectively, a separate equivalent "independent task system" is created to analyze schedulability of each task. When the system consists of partitioned resources, we assume that the transformation described in Section 4 has been performed, and that the priority of jobs is the same on the partitioned resource (within its class) as in other stages of the system that are scheduled in priority order. In Section 5.3, we present DAG schedulability expressions for deadline monotonic scheduling based on the above task set reduction.

5.1 Preemptive Scheduling Transformation

The reduction to a single stage system is conducted by (i) replacing each higher priority job J_i in $\bar{S}_{wc}$ by an equivalent single stage job of execution time equal to $2C_{i,max}(1 + SM_{i,1})$, and (ii) adding a lowest-priority job, J_e^* of execution time equal to $\sum_{j \in Path_1, j \leq N-1} \max_i(C_{i,j})$ (which is the stage-additive component), and deadline same as that of

J_1. By the delay composition theorem, the total delay incurred by J_1 in the acyclic distributed system is no larger than the delay of J_e^* on the uniprocessor, since the latter adds up to the delay bound as expressed in the right hand side of Inequality 1.

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 preempt 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 preempt T_1. If one such invocation preempts 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 preempt T_1 at most once. Extending this argument, at most one invocation of T_i can preempt T_1 at each split-merge between the paths of T_i and T_1. Therefore, the maximum number of preemptions that x invocations of T_i can cause 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 higher priority periodic task T_i by an equivalent single stage task with execution time $C_i^* = 2C_{i,max}$ and having the same period and deadline as T_i, and (ii) adding a lowest priority task T_e^* with computation time $C_e^* = \sum_i C_{i,max} + \sum_i 2C_{i,max}SM_{i,1} + \sum_{j\in Path_1, j\leq N-1} \max_i(C_{i,j})$ (similar to the reduction of the pipelined system as in [10]) with same period and deadline as T_1. If task T_e^* is schedulable on a uniprocessor, so is T_1 on the original acyclic distributed system.

5.2 Non-Preemptive Scheduling Transformation

Under non-preemptive scheduling, we reduce the DAG into an equivalent single stage system under *preemptive* scheduling. This is achieved 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 (2)), 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 (2).

For the case of periodic tasks, an optimization similar to the one described in Section 5.1 can be applied. 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 T_i^* of computation time $C_i^* = C_{i,max}$ and same period and deadline as T_i, and (ii) adding a lowest priority task with computation time $C_e^* = \sum_{i\in\bar{S}_{wc}}(C_{i,max} + C_{i,max}SM_{i,1}) + \sum_{j\in Path_1, j\leq N-1} \max_i(C_{i,j}) + \sum_{j\in Path_1} \max_{i\in M_1(j)} C_{i,max}$ with same period and deadline as T_1.

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.

5.3 Examples of Equivalent Uniprocessor Schedulability Analysis

The reduction described in the previous subsections enables large complex acyclic distributed systems to be easily analyzed using any single stage schedulability analysis technique. For this reason, we call our solution a 'meta-schedulability test'. The only assumptions made by the reduction on the scheduling model are fixed priority preemptive scheduling, and that tasks do not block for resources on any of the stages (i.e., independent tasks). In this section, we show how the Liu and Layland bound [14] and the necessary and sufficient test based on response time analysis [1] can be applied to analyze periodic tasks in an acyclic distributed system, under both preemptive and non-preemptive scheduling. Other uniprocessor schedulability tests can be applied in a similar manner.

Define $C_{i,max}$ as the largest execution time of T_i on any stage, D_i as the end-to-end deadline, and n as the number of periodic tasks in the system. $M_i(j)$ is the set of tasks with lower priority than T_i, whose path merges with T_i at stage j.

For preemptive scheduling, $C_k^* = 2C_{k,max}$; $C_e^*(i) = \sum_{k\geq i} C_{k,max} + \sum_{k>i} 2C_{k,max}SM_{k,i} + \sum_{j\in Path_1, j\leq N-1} \max_{k\geq i}(C_{k,j})$.

For non-preemptive scheduling, $C_k^* = C_{k,max}$; $C_e^*(i) = \sum_{k\geq i}(C_{k,max} + C_{k,max}SM_{k,i}) + \sum_{j\in Path_i, j\leq N-1} \max_{1\leq k\leq n}(C_{k,j}) + \sum_{j\in Path_i} \max_{k\in M_i(j)} C_{k,max}$.

The Liu and Layland bound [14], applied to periodic tasks in an acyclic distributed system is:

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

for each $i, 1 \leq i \leq n$.

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

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

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 should be at most D_i.

6. A Flight Control System Example

In this section, we describe a practical problem faced in flight control systems and explicate how the theory developed in this paper can efficiently solve the problem. In order to keep the example simple and illustrative, we have modified certain attributes of the system. We also show how network scheduling (as a partitioned resource) can easily be handled within the system model assumed in this paper. The purpose of the example is to illustrate how the theory developed in this paper can be applied, and is not intended as a comparison with existing theory on schedulability analysis for distributed systems. Such a comparison is presented in the evaluation section.

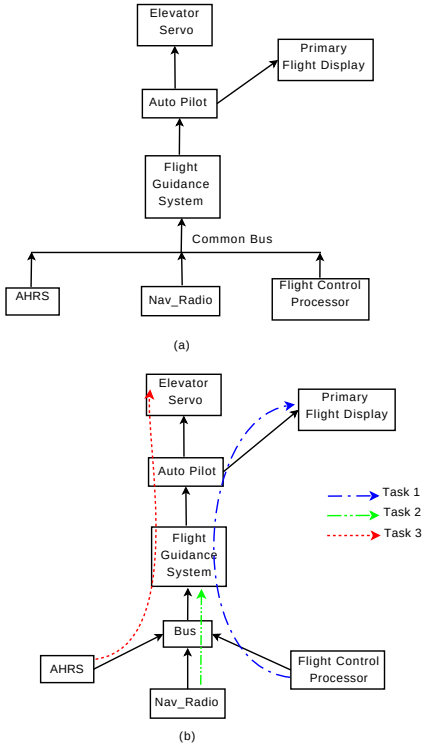

Figure 7. (a) Example flight control system (b) The different flows in the system, with the bus abstracted as a separate stage of execution

A flight control system (with some sub-systems excluded for simplicity) is shown in Figure 7(a). The Flight Guidance System (FGS) receives periodic sensor readings from the Attitude and Heading Reference System (AHRS) and the Navigation Radio (NAV_RADIO). The sensory information gets processed by the FGS and the Auto-Pilot (AP), and the elevator servo component performs the actuation. The Flight Control Processor (FCP) is responsible for input commands from the pilot and display settings. Commands from the FCP need to be processed by the FGS, and display information needs to be transmitted to the Primary Flight Display (PFD). The actual flight control system uses dedicated buses to carry information from one unit to another. However, in order to illustrate how network scheduling can be handled, we assume the presence of a common bus connecting the FGS to the various units that feed into it. Further, we assume a simple TDMA protocol for bus access, which is a common approach to temporal isolation in avionics.

The various tasks that constitute the system are shown in Figure 7(b). Task T_3, the highest priority task, carries periodic sensory information from AHRS to the FGS. The FGS then processes this information, the AP generates commands, and the Servo performs the actuation (adjusts the pitch). Task T_2 carries sensor readings from the NAV_RADIO to the FGS periodically. Commands from the FCP are routed to the PFD through the FGS and AP in task T_1 and is the lowest priority task in the system. T_3 belongs to a separate class on the bus, and T_2 and T_1 belong to a single class. The TDMA protocol on the bus employs a period of 10ms, and allots the first 4ms to the AHRS, the next 6ms to the NAV_RADIO (T_2) and FCP (T_1). Scheduling of tasks at each stage is preemptive and prioritized. Worst case computation times (hypothetical) for the tasks at different stages, their periods and deadlines, are shown in Table 1 (all values in milli-seconds). A hyphen denotes that the task does not execute on the corresponding stage. The value shown for the tasks under 'Bus' denotes the time taken to carry the periodic information on the bus to the FGS.

	T_1	T_2	T_3
AHRS	-	-	10
NAV	-	10	-
FCP	15	-	-
FGS	10	20	15
AP	15	-	20
Servo	-	-	10
PFD	10	-	-
Period	500	250	100
Deadline	450	200	100
Bus	15	6	4

Table 1. Task characteristics (in ms)

Due to space limitations, we analyze schedulability of T_1 only. Schedulability of other tasks can be analyzed similarly. We first need to transform the partitioned bus, into a resource that is scheduled in priority order as described in Section 4. T_1 and T_2 together have a time slot of 6ms every 10ms. The partitioned bus is thus no worse than a dedicated prioritized resource providing service to T_1 and T_2 at a rate slower by a fraction $\frac{6}{10}$, and causing an additional delay of $10-6=4ms$. The computation time of T_1 on the transformed bus can be taken as $15\times\frac{10}{6}+(10-6)=29ms$. The computation time of T_2 on the bus is $6\times\frac{10}{6}=10ms$. From the computation times provided in Table 1, we can obtain $C_{3,max} = C_{2,max} = 20ms$ and $C_{1,max} = 29ms$ (on the bus); $SM_{3,1} = SM_{2,1} = SM_{1,1} = 0$.

As shown in Section 5.1, the reduction for this system scheduled preemptively can be conducted by (i) replacing T_3 and T_2 by equivalent single stage tasks T_3^* and T_2^*, with execution times $C_3^* = 2C_{3,max} = 40ms$ and $C_2^* = 2C_{2,max} = 40ms$, and periods $P_3^* = 100ms$ and $P_2^* = 250ms$; (ii) adding a lowest priority task T_e^* with computation time $C_e^* = C_{3,max} + C_{2,max} + C_{1,max} + \sum_{j=FCP,Bus,FGS,AP} \max_i(C_{i,j})$, i.e., $C_e^* = 20 + 20 +$

$29 + 15 + 29 + 20 + 20 = 153ms$ and having a deadline of 450ms. Applying the response time analysis test [1], we obtain the worst case delay of T_e^* in the single stage system as $393ms$, which is less than the deadline. As T_e^* is schedulable on the hypothetical uniprocessor system, from the delay composition theorem, T_1 is schedulable in the flight control system.

An important requirement in such time-critical systems is to have complete knowledge of dependencies and to be able to determine how changes in the timing properties of one task would affect the schedulability of the system. The analysis developed in this paper can be applied on the fly to check the schedulability of the system when the timing properties of individual tasks change during the design and development of the system.

7. Handling Tasks whose Sub-Tasks Form a DAG

In the discussion so far, we have only considered tasks whose sub-tasks form a *path* in the Directed Acyclic Graph. In this section, we describe how this can be extended to tasks whose sub-tasks themselves form a DAG. We shall refer to such tasks as DAG-tasks. Figure 8(a) shows an example task, whose sub-tasks form a DAG. Edges in the DAG, as before, indicate precedence constraints between sub-tasks and each sub-task executes on a different resource. A sub-task s can execute only after all sub-tasks which have edges to sub-task s have completed execution. In the task shown in the figure, sub-task 5 can execute only after sub-tasks 2 and 3 have completed execution. We call this a 'merger' of sub-tasks. Note that a split, that is, edges from one sub-task s to two or more sub-tasks indicate that once sub-task s completes, it spawns multiple sub-tasks each executing in parallel. It can be observed from the example in Figure 8(a), that once sub-task 1 completes, it spawns sub-tasks 2 and 3 that can execute in parallel on different stages.

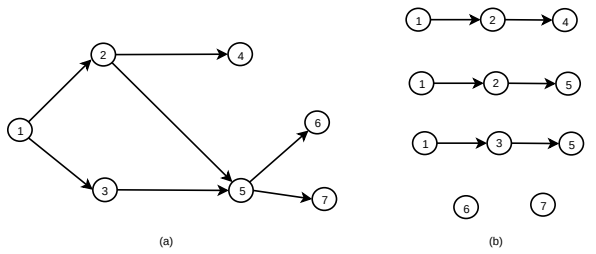

Figure 8. (a) Figure showing an example of a DAG-task (b) Different parts of the DAG-task that need to be separately analyzed to analyze schedulability of the DAG-task.

As the delay composition theorem only addresses tasks which execute in sequential stages (that is, the sub-tasks form a path in the DAG) and does not consider DAG-tasks, we need to break the DAG-task into smaller tasks which form a path of the DAG. This is carried forth as follows. Similar to traditional distributed system scheduling, artificial deadlines are introduced after each merger of sub-tasks. Each split in the DAG creates additional paths that need to be analyzed (the number of additional paths is one less than the fan-out). In the example DAG-task, an artificial deadline is imposed after sub-task 5. Sub-tasks 6 and 7 are analyzed independently using any single stage schedulability test. As there are two splits within sub-tasks 1 through 5, there are 3 paths that need to be analyzed as shown in Figure 8(b). The path 1-2-4 is analyzed independently using the meta-schedulability test and this sequence of sub-tasks need to complete within the end-to-end deadline of the DAG-task. The paths 1-2-5 and 1-3-5 can be independently analyzed using the meta-schedulability test, with their deadline set as the artificial deadline. Sub-tasks 6 and 7 need to complete in a duration at most equal to the end-to-end deadline of the DAG-task minus the artificial deadline set for sub-task 5. If all the parts of the DAG-task are determined to be schedulable, then the DAG task is deemed to be schedulable.

As observed in [10], imposing artificial deadlines add to the pessimism of the schedulability analysis. The use of the delay composition theorem reduces the need to impose artificial deadlines to only stages in the execution where two or more sub-tasks merge. This is in contrast to traditional distributed schedulability analysis, that imposes artificial deadlines after each stage of execution, causing the pessimism to quickly increase with system scale.

8. Simulation Results

In this section, we evaluate the preemptive and non-preemptive schedulability analysis techniques described in Section 5.3. 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 (and due to paucity of space), we consider only periodic tasks in this evaluation. Due to paucity of space, we assume that partitioned resources within the system have been transformed into resources scheduled in priority order as described in Section 4, and focus this evaluation on prioritized resources. An admission controller is used to maintain real-time guarantees within the system. The admission controller is based on a single stage schedulability test for deadline monotonic scheduling, such as the Liu and Layland bound [14] or response time analysis [1], together with our reduction of the multistage distributed system to a single stage, as shown in Section 5.3. 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. When comparing different admission controllers, each admission

controller was allowed to execute on the same 100 task sets.

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 (we do not consider DAG tasks in this evaluation), with each node in the distributed system having a probability of NP (for Node 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 NP is chosen as 0.8. End-to-end 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 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 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 two 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.

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 [17], applied to preemptive scheduling, for different number of nodes in the DAG, the results of which are shown in Figure 9. For meta-schedulability test curves that are marked preemptive, the scheduling was preemptive and the preemptive version of the test was used in admission control. Likewise, for the meta-schedulability test curves that are marked non-preemptive, the scheduling was non-preemptive and the non-preemptive version of the test was used. We only evaluated holistic analysis applied to preemptive scheduling as presented in [17], as the non-preemptive version presented in [12] adds an extra term to account for blocking due to lower priority tasks and tends to be more pessimistic than the preemptive version, and the corresponding curve would always be lower than the curve for preemptive scheduling.

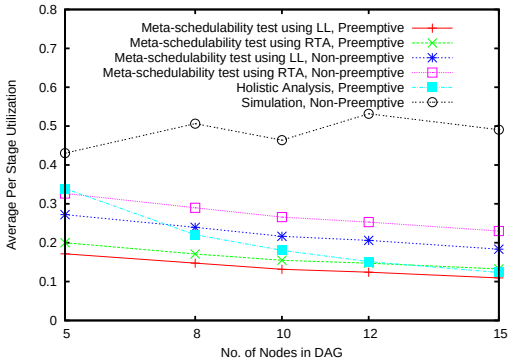

Figure 9. Comparison of meta-schedulability test using both preemptive and non-preemptive scheduling with holistic analysis for different number of nodes in the DAG

It can be observed from Figure 9, that 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. As the utilization curve for holistic analysis applied to non-preemptive scheduling would be lower than the curve for the preemptive scheduling version of holistic analysis, non-preemptive scheduling analyzed using our meta-schedulability test would also outperform the non-preemptive version of holistic analysis. 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.

In order to estimate when deadlines are actually being missed, and to evaluate the pessimism of the admission controllers, we conducted simulations to identify the lowest utilization at which deadlines are missed. The curve labeled 'Simulation' in Figure 9 presents the results from simulations of the lowest utilization at which deadline misses were observed for different number of nodes in the system when non-preemptive scheduling was employed. The corresponding curve for preemptive scheduling, was within 0.02 of those of non-preemptive scheduling, and we don't show the values here for the sake of clarity (the reader must bear in mind

that task sets were generated randomly, and that the task sets do not represent worst case scenarios). Each point for the simulation curve was obtained from 500 executions of the simulator in the absence of any admission controller, with each execution considering a workload with utilization close to where deadline misses were being observed. We observe that the lowest utilization at which deadline misses were observed does not decrease with increasing system scale. The meta-schedulability test curves degrade only marginally with increasing scale, while the performance of holistic analysis degrades more rapidly.

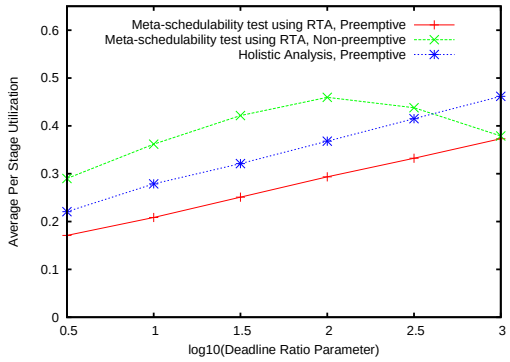

Figure 10. Comparison of meta-schedulability test using both preemptive and non-preemptive scheduling with holistic analysis for different deadline ratio parameters

To precisely evaluate the scenarios under which non-preemptive scheduling performs better than preemptive scheduling in distributed systems, we conducted experiments varying the deadline ratio parameter (DR) while keeping the other parameters equal to their default values. Figure 10 plots a comparison of the meta-schedulability test under both preemptive as well as non-preemptive scheduling, with holistic analysis for different DR values ranging between 0.5 and 3.0. A 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. Up to $DR = 2$, non-preemptive scheduling (together with the non-preemptive version of the meta-schedulability test) results in better performance than preemptive scheduling (together with the preemptive version of the test). 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 achievable utilization under non-preemptive scheduling decrease beyond a DR value of 2, as higher priority tasks can now be blocked for a longer duration under non-preemptive scheduling, leading to a greater likelihood of deadline misses.

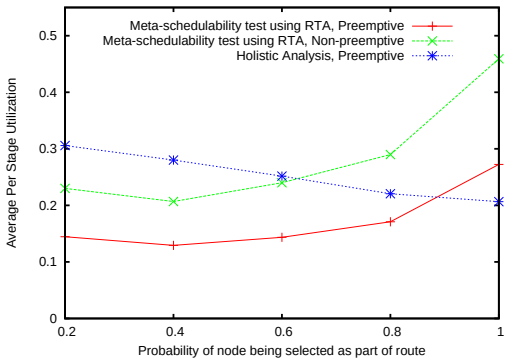

Figure 11. Comparison of meta-schedulability test using both preemptive and non-preemptive scheduling with holistic analysis for different route probabilities

We conducted a similar comparison of the three admission controllers as in the previous experiment, but for different values of the Node Probability (NP) parameter, which is the probability with which each node in the system is chosen as part of the route of each task. This comparison is shown in Figure 11, for different NP parameter values ranging between 0.2 to 1.0 in steps of 0.2. Note that the NP 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 NP, the number of stages on which each task executes is low, and as observed in Figure 9, holistic analysis performs better than the meta-schedulability test. However, for larger values of NP, 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 NP values greater than 0.6.

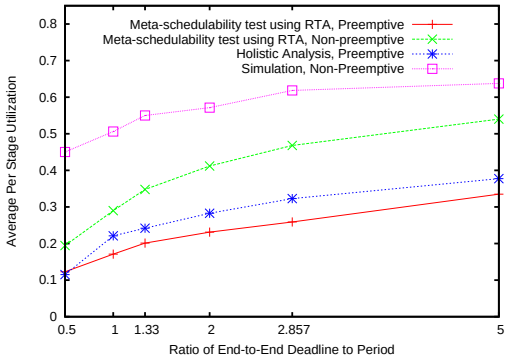

Figure 12. Comparison of meta-schedulability test using preemptive and non-preemptive scheduling with holistic analysis for different ratios of end-to-end deadline to task periods

The above results have all been obtained by setting the end-to-end deadlines equal to the periods of tasks. Figure 12 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 scheduling consistently outperforms preemptive scheduling analyzed using either the meta-schedulability test or holistic analysis. As holistic analysis applied to non-preemptive scheduling (curve not shown) would perform worse than the preemptive scheduling version of holistic analysis, it would also perform worse than the meta-schedulability test applied to non-preemptive scheduling. Similar to Figure 9, the curve labeled as 'simulation' plots the lowest utilization at which deadline misses were observed obtained from simulations under non-preemptive scheduling in the absence of any admission controller. The corresponding values for preemptive scheduling were close to those obtained for non-preemptive scheduling and are not presented here for the sake of clarity. We observe that our analysis tends to be less pessimistic for larger values of the ratio between the end-to-end deadline and the period.

9. Related Work

In their seminal work [14], Liu and Layland 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, 13] were also proposed.

Algorithms for statically scheduling precedence constrained tasks in distributed systems have been proposed in [18, 6]. Such algorithms construct a schedule of length equal to the least common multiple of the task periods of the set of periodic tasks. This schedule can then be used to accurately specify the time intervals during which each task will be executed. Such algorithms have a huge time complexity and are clearly unsuitable for large, complex distributed systems.

A few offline schedulability tests have also been proposed, which 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]. In [15, 16], offset-based response time analysis techniques for EDF were proposed, which divide the end-to-end deadline into individual stage deadlines. Recently, a middleware layer based on deferrable servers for aperiodic tasks with hard end-to-end deadlines in distributed real-time applications was designed and implemented in [19]. Techniques to divide the end-to-end deadline into sub-deadlines for individual stages were presented.

Holistic schedulability analysis for distributed hard real-time systems [17], assumes the worst case delay at a stage as the jitter for the next stage. While this technique does not divide the end-to-end deadline into sub-deadlines for individual stages, it nevertheless does not account for the overlap in the execution of different pipeline stages.

In [7], a schedulability test based on aperiodic scheduling theory for fixed priority scheduling was derived. Although this solution handles arbitrary-topology resource systems and resource blocking, it does not consider the overlap in the execution of multiple stages in the system. In an earlier publication [10], we proved a delay composition theorem for pipelined systems. This paper extends the results in [10] to directed acyclic systems and non-preemptive scheduling.

In stark contrast to preemptive scheduling, non-preemptive scheduling has received very little attention from the real-time community. An extension to holistic analysis in distributed systems to account for blocking due to non-preemptive scheduling is presented in [12]. The paper presents a comparison of this analysis technique with network calculus [5], 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. In contrast to such techniques, we reduce the problem of analyzing a distributed acyclic system with non-preemptive scheduling to that of analyzing a single stage system using preemptive scheduling. Thus, well known uniprocessor tests can be adopted to analyze multistage systems that use non-preemptive scheduling, resulting in more efficient schedulability analysis.

10. Conclusions and Future Work

In this paper, we present a delay composition theorem that bounds the worst-case delay of jobs for preemptive and non-preemptive scheduling in distributed systems, where the routes of tasks form a directed acyclic graph. We consider systems where resources can be either partitioned or scheduled in priority order. The rule works purely based on information available along the route followed by the task under consideration, and does not require global knowledge of other parts of the distributed system. The composition rule leads to the reduction of the distributed system to an equivalent single stage system, which then enables any single stage schedulability test to analyze distributed systems. We show that under certain conditions, non-preemptive scheduling can perform better than preemptive scheduling for distributed systems. We believe this observation can foster more extensive study and use of non-preemptive scheduling in distributed systems.

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 derived in this paper can help in the search for an 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. Extensions to allow both preemptive and non-preemptive scheduling to be employed within the same system, will also be useful.

References

[1] A. N. Audsley, A. Burns, M. Richardson, and K. Tindell. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering*, 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 *13th ECRTS*, 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 *ICDCS*, pages 204–211, May 1995.

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

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

[7] W. Hawkins and T. Abdelzaher. Towards feasible region calculus: An end-to-end schedulability analysis of real-time multistage execution. In *IEEE RTSS*, December 2005.

[8] P. Jayachandran and T. Abdelzaher. The case for non-preemptive scheduling in distributed real-time systems. Technical Report UIUCDCS-R-2007-2852, University of Illinois at Urbana-Champaign, IL, USA, 2007.

[9] P. Jayachandran and T. Abdelzaher. A delay composition theorem for real-time distributed directed acyclic systems. Technical Report UIUCDCS-R-2007-2851, University of Illinois at Urbana-Champaign, IL, USA, 2007.

[10] P. Jayachandran and T. Abdelzaher. A delay composition theorem for real-time pipelines. In *Euromicro Conference on Real-Time Systems (to appear)*, July 2007.

[11] P. Jayachandran and T. Abdelzaher. Transforming distributed acyclic systems into equivalent uniprocessors under preemptive and non-preemptive scheduling. Technical report, University of Illinois at Urbana-Champaign, IL, USA, 2007.

[12] 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*, 42(14):2899–2913, July 2004.

[13] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *RTSS*, pages 166–171, December 1989.

[14] 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.

[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] R. Pellizzoni and G. Lipari. Improved schedulability analysis of real-time transactions with earliest deadline scheduling. In *RTAS*, pages 66–75, March 2005.

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

[18] 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.

[19] 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.