End-to-End Delay Analysis of Distributed Systems with Cycles in the Task Graph

Praveen Jayachandran and Tarek Abdelzaher
Department of Computer Science, University of Illinois at Urbana-Champaign, IL 61801

Abstract

A significant problem with no simple solutions in current real-time literature is analyzing the end-to-end schedulability of tasks in distributed systems with cycles in the task graph. Prior approaches including network calculus and holistic schedulability analysis work best for acyclic task flows. They involve iterative solutions or offer no solutions at all when flows are non-acyclic. This paper demonstrates the construction of the first generalized closed-form expression for schedulability analysis in distributed task systems with non-acyclic flows. The approach is a significant extension to our previous work on schedulability in Directed Acyclic Graphs. Our main result is a bound on end-to-end delay for a task in a distributed system with non-acyclic task flows. The delay bound allows one of several schedulability tests to be performed. Evaluation shows that the schedulability tests thus constructed are less pessimistic than prior approaches for large distributed systems.

1. Introduction

Real-time applications are becoming increasingly complex in terms of system scale and the number of resources involved. With Moore’s Law approaching saturation, emphasis shifts towards increasing distribution. Elegant uniprocessor solutions need to be generalized to distributed environments. Towards that goal, in previous publications, the authors explored delay composition in pipelines [5] and distributed systems of directed acyclic task graphs (DAGs) [8]. An algebra was described for reducing such DAGs to equivalent unprocessors [6] that can then be analyzed using existing uniprocessor schedulability tests.

This paper significantly extends the scope of applicability of past results by introducing the first reduction-based schedulability analysis technique that applies to distributed systems with non-acyclic task graphs. Informally, a task graph is non-acyclic if task flows in the underlying distributed system include cycles. Most common types of traffic do, in fact, have non-acyclic behavior. For example, request-response traffic in client-server systems includes flows (of requests) from client machines to server machines and flows (of responses) in the reverse direction. Hence, analysis of end-to-end client-side latency entails analysis of a non-acyclic task flow. Reliability mechanisms that entail transmission and processing of acknowledgments, as well as token passing mechanisms are other examples of systems with non-acyclic task flows.

The fundamental problem in handling task graphs that contain cycles is that the arrival pattern of jobs to a particular node in the system is directly or indirectly dependent on the rate at which jobs exit the node downstream, but that downstream pattern is in turn dependent on the load of the node under consideration and hence on this node’s arrival pattern. This is a cyclic dependency. A common way to break this dependency is to impose artificial deadlines on node boundaries, thereby converting the original distributed system problem into a set of uniprocessor schedulability tests, one per node, that ensure that local deadlines are satisfied. The artificially-introduced intermediate deadlines, however, constitute additional requirements not present in the original problem formulation, thereby reducing schedulability and leading to pessimistic solutions. Another common approach is to introduce non-work-conserving behavior such as the use of per-node buffering or traffic regulation. For example, regardless of when a periodic task finishes execution on one node, it may not be considered for scheduling at the next node until the beginning of its next period. This independence from actual completion times breaks the cyclic dependency. The approach yields good results when the end-to-end deadline of the task is of the order of the product of its period and the number of nodes traversed. It does not work well when tasks have end-to-end deadlines that are, for example, smaller than their period.

Existing techniques to analyze end-to-end delay in distributed systems, without introducing artificial deadlines or relying on non-work-conserving behavior, include network calculus [2, 3] and holistic analysis [17], together with various extensions such as [15]. They can accommodate buffering but do not require it. These techniques analyze the system one node at a time. Given information regarding the arrival pattern of jobs entering a particular node, they analyze the execution on the node to determine information regarding the exit pattern of jobs leaving the
node, which in turn becomes the arrival pattern of jobs to a future node. In the absence of cycles in the task graph, one can always find a partial order, in applying per-node analysis, such that all the required information with regard to the arrival pattern of jobs to a particular node (in terms of jitter and offset information for holistic analysis techniques, or arrival curve information for network calculus) is available when analyzing that node. However, when the task graph contains cycles, this technique breaks down due to the cyclic dependency. To overcome this problem, an iterative procedure is suggested in prior literature [14, 15] and is shown to converge, but the process becomes quite complicated when handling many tasks and nodes.

With the principal focus of being able to easily analyze end-to-end delay and schedulability of tasks in systems that contain cycles in the task graph, in this paper, we derive a simple expression for a worst-case task delay bound in non-acyclic systems, cast in terms of the computation times of higher priority jobs in the system. The result provides a natural means of handling loops and does not incur additional complexity, requirements, or non-work-conserving delays when the task graph contains cycles, unlike existing analysis techniques. By considering the system as a whole rather than analyzing it one node at a time, the bound accurately accounts for the concurrency in the execution of different nodes, resulting in a less pessimistic bound on the end-to-end delay. This is especially valuable when the system is large and when deadlines are short (e.g., not much longer than periods).

Our expression for the delay bound does not rely on periodicity assumptions and hence is applicable to periodic as well as aperiodic scheduling. The only assumption made is that a job has the same relative priority across all nodes on which it executes. With this stipulation, the bound is applicable to both static and dynamic priority scheduling, as it imposes no constraints on priorities among different jobs. We provide a simple and intuitive proof for the delay bound under preemptive scheduling. We only state the version of the bound under non-preemptive scheduling and omit the proof as it is similar.

Being a reduction-based approach to schedulability analysis [8], the derived end-to-end delay bound provides a means by which the problem of analyzing schedulability of tasks in a distributed system with cycles can be reduced to that of analyzing schedulability in an equivalent hypothetical uniprocessor. Thus, well-known uniprocessor analysis techniques can be used to analyze the schedulability of tasks in arbitrary distributed systems.

We envision that this new technique to analyze task graphs that contain cycles will lend itself towards developing a general theory of understanding timing behavior in distributed systems. While we have a clear understanding of which scheduling policies perform well for unprocessors, there has been little work done in analyzing optimal (or good) scheduling policies or optimizing priority assignment for distributed tasks. As the end-to-end delay bound is estimated after the priorities of jobs have been assigned, it can be used to optimize priority assignment based on different metrics such as minimizing end-to-end delay or maximizing the system utility. Other design issues such as how to optimally route tasks within the system, require an in-depth understanding of how system topology affects end-to-end delay. We hope that future work will address these critical issues and provide us with a better understanding of timing behavior in distributed systems.

The rest of this paper is organized as follows. We review related work in Section 2. We describe the system model in Section 3 and state and prove the end-to-end delay bound for jobs in non-acyclic systems in Section 4. In Section 5 we briefly describe how the end-to-end delay bound can be used to reduce the schedulability problem of tasks in distributed systems to that of analyzing an equivalent hypothetical uniprocessor. We illustrate the advantage of using the analysis technique presented in this paper using an example in Section 6. In Section 7, we present simulation studies and conclude the paper in Section 8.

2. Related Work

The problem of analyzing the schedulability of tasks in distributed systems has been studied using different techniques in the past. However, these techniques become more pessimistic or provide no solutions when the task graph contains cycles and when the system size is large. We lack a clear understanding of how cycles in the task graph affect the end-to-end delay and schedulability of tasks.

Algorithms such as [19, 4] have been proposed to statically schedule precedence constrained tasks in distributed systems. A schedule of length equal to the least common multiple of the task periods is constructed, that would precisely define the time intervals of execution of each job. Clearly, these algorithms have exponential time complexity and are not suited for large distributed systems.

Offline schedulability tests have been proposed that divide the end-to-end deadline of tasks into per-stage deadlines. The end-to-end task is then considered as several independent sub-tasks, each executing on a single stage in the system. Uniprocessor schedulability tests are then used to analyze if each stage is schedulable. If all the stages are schedulable, the system is deemed to be schedulable. We refer to this technique as traditional in our simulation studies. For instance, [10, 20] present techniques to divide the end-to-end deadline into per-stage deadlines. While this technique does not incur any problems with handling cycles in the task graph, it tends to be extremely pessimistic and does not accurately account for the inherent parallelism in the execution of different stages. A technique that combines offline and online scheduling is proposed in [13]. 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 analysis was first proposed in [17], and has since had several extensions such as [14, 15] that propose offset-based response time analysis techniques for EDF. In addition to the computation time and period, tasks are characterized by two other parameters, namely the jitter and the offset. The jitter and offset information is used to characterize the arrival pattern of tasks to each stage in the distributed system. The fundamental principle behind holistic analysis and its extensions is that, given the jitter and offset information of jobs arriving at a stage one can compute (in a worst-case manner) the jitter and offset for jobs leaving the stage, which in turn becomes the arrival pattern for jobs to a subsequent stage. By successively applying this process to each stage in the system, one can compute a worst-case bound on the end-to-end delay of jobs. However, this technique works only in the absence of cycles in the task graph. In the presence of cycles, the jitter and offset of jobs at a stage (that is part of the cycle) becomes directly or indirectly dependent on the jitter and offset of jobs leaving the stage, resulting in a cyclic dependency. To overcome this problem, an iterative procedure is described in [14, 15] which is shown to converge. This solution technique, however, becomes tedious, complicated and quite pessimistic for large task graphs with tens of nodes.

From the networking perspective, network calculus [2, 3] was proposed to analyze the end-to-end delay of packets of flows. This was applied to the context of real-time systems, called Real-Time Calculus first presented in [16], and has since been extended to handle different system models such as [9, 18]. In approaches based on network calculus, the arrival pattern of jobs of flows is characterized by an arrival curve. Given a service curve for a node based on the scheduling policy used, one can determine the rate at which jobs leave the node after completing execution, which in turn serve as the arrival curve for the next stage in the flow’s path. For task graphs that contain cycles, we are faced with the same cyclic dependency problem. In [3], a general solution to this problem is presented by setting up a system of simultaneous equations, which becomes difficult or impossible to solve for large systems. Moreover, the equations tend to be non-linear, and even if linearizing the equations is possible, it can be a significant source of pessimism. Needless to say, there is no means by which the solution can be efficiently automated for arbitrary task graphs. A comparison of holistic analysis and network calculus was conducted in [11], where holistic analysis was found to be less pessimistic than network calculus in general. We show in the evaluation section that both of these techniques tend to become increasingly pessimistic with system scale. In contrast, in this paper, we derive a simple bound on the end-to-end delay of a job in terms of the computation times of higher priority jobs that can delay it. It accurately accounts for parallelism in the execution of different stages, resulting in a less pessimistic estimate of the end-to-end delay.

A delay composition theorem that bounds the worst-case end-to-end delay of jobs in pipelined systems under preemptive and non-preemptive scheduling was derived in [5, 7]. This was extended to directed acyclic graphs, and also to partitioned resources (e.g., TDMA scheduling) in [8]. In this paper, we provide an end-to-end delay bound for tasks in distributed systems that contain cycles in the task graph.

3. System Model

Our model of non-acyclic distributed processing consists of a distributed system of \(N\) nodes and a set of real-time jobs. These jobs may or may not be instances of periodic tasks. Our proof techniques do not rely on periodicity assumptions. Each node is a resource, which is anything that is allocated to jobs in priority order. For instance, the resource could be a processor or a point-to-point communication link. A given job, \(J_k\), has the same relative priority across all resources in the distributed system. Different jobs require processing at a different sequence of nodes in the distributed system, and may have different start and end nodes.

Since jobs may revisit nodes, it is useful to differentiate between nodes and stages visited by a job. A stage is simply an instance of visiting a node. For example, a job that visits nodes 1, 2, then 1 is said to have a sequence of three stages, during which it visits the aforementioned nodes. Let the sequence of stages traversed by job \(J_k\) be called its path, \(p_k\). In a departure from our previously published models [5, 8, 6], the union of paths traversed by all jobs may contain loops. For example, a job can revisit a node, or two jobs can visit two nodes in different orders. We therefore say that the path of job \(J_k\) contains one or more folds. A fold of \(J_k\) starting at node \(i\) is the largest sequence of nodes (in the order traversed by job \(J_k\)) that does not repeat a node twice. The first fold on path \(p_k\) starts with the first node that \(J_k\) visits. We denote the \(x^{th}\) fold of job \(J_k\) by \(J^x_k\). For instance, if \(J_k\) has the path \((1, 2, 3, 1, 5, 6, 2)\), it is said to have two folds, namely \((1, 2, 3)\) and \((1, 5, 6, 2)\), denoted by \(J^1_k\) and \(J^2_k\) respectively. If the path of a job is acyclic, then it has only one fold that contains the whole path. The intuition for defining folds is that when jobs revisit a node multiple times, they may delay other jobs more than once on the same stage. In contrast, a single fold (of a job) can delay other jobs at most once per stage. Hence, folds will simplify the presentation of our proof. We denote the set of all folds of job \(J_k\) by \(Q_k\).

Each job \(J_k\) must complete execution on all stages along its path \(p_k\) within its prespecified end-to-end deadline. The union of all the job paths forms a task graph. An arc in the task graph represents the direction of execution
flow of a job, yielding a precedence constraint between the execution of the sub-jobs at the head and tail nodes of the arc. Observe that the task graph may contain cycles even if all jobs had one fold each. For example, consider a system of two jobs that traverse a sequence of nodes in opposite directions, such as the one shown in Figure 1. The task graph for this system contains a loop, as shown in figure, even though individual jobs don’t. Hence, loops in the task graph capture cyclic dependencies that may involve one or more jobs.

**Figure 1. An task graph with a cycle**

Let \( C_{k,j} \) denote the worst-case execution time of job \( J_k \) on stage \( j \) in its path (simply called execution-time or stage execution time in the rest of this paper), and let \( D_i \) denote the relative end-to-end deadline of job \( J_k \). The problem addressed in this paper is to determine whether or not all deadlines are met.

4. Delay in Non-Ayclic Task Graphs

In this section, we present the derivation of a worst-case end-to-end delay bound for a job in a distributed system with loops in the task graph. This derivation enables construction of compact schedulability tests to determine if the system is schedulable. The worst-case end-to-end delay bound for a job is expressed in closed-form in terms of the computation times of higher priority jobs that can preempt or delay it.

Let all jobs be numbered in priority order such that larger integers denote higher priority. When analyzing the delay of a job, since scheduling is preemptive and there is no blocking in our model, lower priority jobs can be ignored. Hence, without loss of generality, let the job whose end-to-end delay we wish to bound be denoted by \( J_1 \). This job executes along a path \( p_1 \) in the distributed system, where \( p_1 \) may contain one or more folds.

We ignore the precedence constraints between successive folds of each higher priority job \( J_i \), where \( i > 1 \). Thus, each fold of a higher priority job \( J_i \) becomes an independent job. We call the \( x \)th fold of job \( J_i \) by (job) \( J_i^x \). We call this process unfolding. Observe that unfolding does not eliminate cycles in the task graph because different folds of the same or different jobs can still visit nodes in different orders. Unfolding ensures, however, that job \( J_1 \) is delayed by any one fold (of a higher-priority job) at most once per \( J_1 \)’s stage.

It is easy to show that unfolding cannot decrease the delay of job \( J_1 \). Hence, if \( J_1 \) is schedulable after unfolding, then it is schedulable in the original job set. This is because unfolding merely removes some of the (precedence) constraints between stages of higher priority jobs. Hence, it increases the set of feasible higher-priority task arrival patterns that one needs to consider. A bound on \( J_1 \)’s delay computed by maximization over the larger set of possible arrival patterns can only be larger than one computed by maximization over the subset that respects the removed constraints (thus erring on the safe side). In the following, we therefore consider the unfolded job set when analyzing the delay of \( J_1 \).

Note that, a fold \( J_1^x \) can only preempt or delay \( J_1 \) when it shares a common execution node or a common sequence of nodes with \( J_1 \). Let us define a job segment \( J_i^{x_1,x_2} \) as \( J_i^x \)’s execution on a sequence of consecutive nodes on the path of \( J_i^x \) that is also traversed by \( J_1 \) either in the same order or exactly in reverse order. Let \( Seg_i^x \) be the set of all such segments for \( J_i^x \). For example, if \( J_1 \) has the path \( (1,2,1,3,8,11,13) \) and \( J_i^x \) has the path \( (1,3,19,13,11,8) \), then \( Seg_i^x = \{ J_i^{x,1}, J_i^{x,2} \} \), where \( J_i^{x,1} \) is the part of \( J_i^x \) that executes on nodes \((1,3)\), and \( J_i^{x,2} \) is the part of \( J_i^x \) that executes on nodes \((13,11,8)\).

Consider \( J_1 \) and the job segments (segments for short) that delay or preempt its execution. Each such segment falls in one of three categories:

- **Forward flow segments**: Those are segments that share a consecutive set of stages with \( J_1 \) and traverse them in the same direction.
- **Reverse flow segments**: Those are segments that share a consecutive set of stages with \( J_1 \) and traverse them in the opposite direction.
- **Cross flow segments**: Those are segments composed of only one node. For example, such a segment may result from intersection of the path of \( J_1 \) with the path of another job in one node.

Figure 2 shows an example where the path of \( J_1 \) traverses five stages. Higher-priority job segments that share parts of that path are indicated by arrows that extend across the stages they execute on, pointing in the direction of the flow of the segment. Cross-flow segments are indicated by vertical arrows at the node they execute on.
last stage. The length of this interval is the end-to-end response time of $J_1$, which we wish to bound. Let us now define a busy execution trace, to mean a sequence of contiguous intervals of continuous processing on successive stages of path $p_1$ that collectively add up to the end-to-end delay of $J_1$. The intervals are contiguous in the sense that the end of a processing interval on one stage is the beginning of another processing interval on the next stage of path $p_1$. There may be many execution traces that satisfy the above definition. To reduce the number of different possibilities we further constrain the definition by requiring that each processing interval in the trace end at a job boundary (i.e., when some job’s execution on that stage ends, which we shall call the job’s finish time at that stage). Hence, the definition of a busy execution trace is as follows:

Definition 1: A busy execution trace through path $p_1$ is a sequence of contiguous intervals starting with the arrival of $J_1$ on stage 1 and ending with the finish time of $J_1$ on the last stage of $p_1$, where (i) each interval represents a stretch of continuous processing on one stage, $j$, of path $p_1$, (ii) the interval on stage $j$ ends at the finish time of some job on stage $j$, (iii) successive intervals are contiguous in that the end time of one interval on stage $j$ is the start time of the next interval on stage $j + 1$, and (iv) successive intervals execute on consecutive stages of path $p_1$.

Figure 3 presents examples of execution traces. In this figure, $J_1$, whose execution is indicated in black, traverses four stages, while being delayed and preempted by other jobs. The arrival time of $J_1$ to the first stage and its finish time on the last stage are indicated by $J_1$ in and $J_1$ out, respectively. Traces are depicted as staircase lines where the horizontal parts represent busy intervals at successive stages of $p_1$, and the vertical parts represent traversals to the next stage. Trace A and Trace B, in the figure, are examples of valid busy execution traces by our definition. Trace C does not satisfy the definition because it ends (i.e., runs into idle time) before the finish time of $J_1$ on the last stage. Remember that a busy execution trace, by definition, cannot contain idle time, since it is composed of contiguous intervals of continuous processing, ending with the finish time of $J_1$ on its last stage. In the following, we shall bound the length of a valid busy trace, hence, bounding the end-to-end response-time of $J_1$.

Observe that given work-conserving scheduling on all nodes, at least one busy execution trace always exists. Namely, it is the trace composed of the waiting intervals of $J_1$ on successive stages. This trace is indicated by Trace A in Figure 3. We call this trace the trace of last traversal because it ends its intervals on each stage at the finish time of the last (i.e., lowest priority) job. Let us now define the trace of earliest traversal as follows.

Definition 2: A trace of earliest traversal is a busy execution trace in which the end of an interval on stage $j$ coincides with the finish time of the first job segment on stage $j$ that (i) moves on to stage $j + 1$ next, and (ii) shares at least one future stage $k > j$ with $J_1$, where both execute in the same busy period (or is $J_1$).

The second condition in the definition prevents construction of invalid traces, such as Trace C in Figure 3, that run into idle time before the completion of $J_1$ on the last stage. Because of that condition, one can show by induction that if starting at the first stage, there exists any valid execution trace from the current point on (which is always the case), then no stage traversal in the earliest traversal trace leads to a point that invalidates that property. Consequently, the trace of earliest traversal is always a valid trace.

Bounding the end-to-end delay of $J_1$ is equivalent to bounding the length of the trace of earliest traversal. First, we bound the amount of execution time that each types of job segments may contribute to the earliest traversal trace. For the purpose of expressing the aforementioned bound in a compact manner, it is convenient at this point to define $C_{i,max}^{x,s}$ to denote the maximum single-stage execution time of segment $J_i^{x,s}$ over its joint path with $J_1$, and define $Node_{i,max}$ to denote the maximum stage execution time of all job-segments $J_i^{x,s}$ on node $i$. The three lemmas below bound delays due to the three types of segments depicted in Figure 2; namely, the forward flow segments, reverse flow segments and cross segments. We start with the most obvious ones first.

Lemma 1: A cross-flow segment, $J_i^{x,s}$, contributes at most one stage computation time to the length of the earliest traversal trace (bounded by $C_i^{x,s}$).

Proof. The lemma is trivially true since cross traffic segments, by definition, have only one stage.

Lemma 2: A reverse-flow segment, $J_i^{x,s}$, contributes at most one stage computation time to the length of the earliest traversal trace (bounded by $C_i^{x,s}$).

Proof. The lemma is true because reverse flow segments execute on the nodes of the system in the reverse order.
from $J_1$. Since the earliest traversal trace follows the path of $J_1$, if $J_{1}^{x,s}$ was included in the interval of the trace at stage $j$, then it must have departed stage $j + 1$ before the beginning of the interval of the trace on stage $j + 1$. Similarly, it will arrive at stage $j - 1$ after the end of the interval of the trace on stage $j - 1$.  

**Lemma 3:** The total contribution of all forward-flow segments, $J_{i}^{x,s}$, to the length of the earliest traversal trace is bounded by:

$$\sum_{\text{segments}} C_{i,\text{max}}^{x,s} + \sum_{\text{forward-flow segments}} C_{i,\text{max}}^{x,s} + \sum_{j \in p_1} \text{Node}_{j,\text{max}}$$  

(1)

**Proof.** Let us define the *end stage* of a forward-flow job segment as either its last stage or the stage after which it is always separated by idle time from $J_1$ (and hence need not be considered further), whichever comes first. It is convenient to partition the contribution of forward-flow segments to the length of the trace into (i) the total length due to stage execution times of segments at their end stages, denoted by $C_{ff_1}$, (ii) the total length due to stage execution times of segments not at their end stages, whose finish time is in the trace, denoted by $C_{ff_2}$, and (iii) the total length of stage execution times of segments not at their end stages, whose finish time is not in the trace, denoted by $C_{ff_3}$.

To bound $C_{ff_1}$, the total length due to stage execution times of segments at their end stages, note that each forward-flow segment, $J_{i}^{x,s}$, has only one end stage. Its length is at most $C_{i,\text{max}}^{x,s}$. The total of all end-stage computation times over all segments is thus given by:

$$C_{ff_1} = \sum_{\text{forward-flow segments}} C_{i,\text{max}}^{x,s}$$  

(2)

To bound $C_{ff_2}$, observe that, by definition of the earliest traversal trace (see Definition 2), its interval on stage $j$ terminates at the finish time of the first forward-flow segment on stage $j$ that is not at its end stage. Since the first such forward-flow segment that finishes terminates an interval, there can only be one such segment finishing per interval, thus contributing at most $\text{Node}_{j,\text{max}}$ to the interval. Since the number of intervals in the trace is equal to the number of stages on path $p_1$, the total length contributed by stage execution times of segments not at their end stages, whose finish time is in the trace is bounded by:

$$C_{ff_2} = \sum_{j \in p_1} \text{Node}_{j,\text{max}}$$  

(3)

To bound $C_{ff_3}$, note that, segments not at their end stages, whose finish time is not in the trace can belong to an interval of the trace only if they were, at some point, preempted by another (forward-flow) segment that finished first (thus terminating the interval). The number of such unfinished forward-flow segments is equal to the number of preemptions, each contributing at most $C_{i,\text{max}}^{x,s}$.

Observe that one segment can preempt another segment at most once. This is because after the first preemption, the two segments either continue executing in priority order on their remaining stages or part ways. The total number of preemptions is thus bounded by the total number of segments, contributing an extra amount of execution time equal to:

$$C_{ff_3} = \sum_{\text{segments}} C_{i,\text{max}}^{x,s}$$  

(4)

Adding up $C_{ff_1}$, $C_{ff_2}$ and $C_{ff_3}$, given by Equations (2), (3), and (4), the lemma follows.  

Consider all jobs $J_i$, each made of a set of folds, denoted by $Q_i$, where each fold $J_{i}^{x} \in Q_i$ gives rise to one or more segments, $J_{i}^{x,s}$, collectively called set $\text{Seg}_{i}^{x}$. The following theorem presents the delay bound on $J_1$ in the system.

**Theorem 1.** For a preemptive, work-conserving scheduling policy that assigns the same priority across all stages for each job, and a different priority for different jobs, the end-to-end delay of a job $J_1$ following path $p_1$ can be composed from the execution parameters of higher priority jobs that delay or preempt it as follows:

$$\text{Delay}(J_1) \leq \sum_{i} \sum_{J_{i}^{x} \in Q_i} \sum_{J_{i}^{x,s} \in \text{Seg}_{i}^{x}} 2C_{i,\text{max}}^{x,s} + \sum_{j \in p_1} \text{Node}_{j,\text{max}}$$  

(5)

**Proof.** The theorem follows trivially from Lemma 1, 2, and 3, by adding the contributions of all cross-flow, reverse-flow, and forward-flow segments to the trace.  

We shall now state the theorem under non-preemptive scheduling, but omit its proof in the interest of brevity. Let $\text{Node}_{j,\text{all, max}}$ denote the maximum computation time of any job (not just higher priority jobs) on stage $j$, and let $\text{Node}_{j,\text{lower, max}}$ denote the maximum computation time of any lower priority job that joins the path of $J_1$ on stage $j$.

**Theorem 2.** For a non-preemptive scheduling policy that assigns the same priority across all stages for each job, and a different priority for different jobs, the end-to-end delay of a job $J_1$ following path $p_1$ can be composed from the execution parameters of jobs that delay it as follows:

$$\text{Delay}(J_1) \leq \sum_{i} \sum_{J_{i}^{x} \in Q_i} \sum_{J_{i}^{x,s} \in \text{Seg}_{i}^{x}} C_{i,\text{max}}^{x,s} + \sum_{j \in p_1} \text{Node}_{j,\text{all, max}}$$

$$+ \sum_{j \in p_1} \text{Node}_{j,\text{lower, max}}$$  

(6)
5. Schedulability Analysis

Using the end-to-end delay bound for non-acyclic systems derived in the previous section, we can reduce the schedulability analysis of tasks in a distributed system with cycles to that of analyzing an equivalent hypothetical uniprocessor, similar to the technique presented in [8]. To analyze the schedulability of a job \( J_1 \), the transformation is carried forth as follows:

- Each higher priority job-segment \( J^z_{i,s} \) in the distributed system, is replaced by a uniprocessor job \( J^z_{i,s} \) with computation time equal to \( 2C_{i,\text{max}} \) and same deadline as \( J_1 \);

- Job \( J_1 \) is replaced by a uniprocessor job \( J^*_1 \) with computation time equal to \( C_{1,\text{max}} + \sum_{j \in p_i} N_{j,\text{max}} \) and deadline same as \( J_1 \).

Hence, if the uniprocessor job \( J^*_1 \) is schedulable, so is job \( J_1 \) in the original distributed system. In the case of periodic tasks, uniprocessor jobs which are invocations of the same periodic task can be grouped together to form a periodic task on the uniprocessor. When the end-to-end deadlines of tasks are larger than the period, then for each higher priority task \( T_i \) we need to account for the task invocations that can be present in the system when \( J_1 \) arrives, which can be bounded by \( [D_i/P_i] \). Further, if the task \( T_i \) being analyzed has cycles in its path, then earlier invocations of \( T_i \) may delay invocations that arrive later. Therefore, \( T_1 \) also needs to be included in the set of higher priority tasks. When the end-to-end deadline of tasks is lesser than the period, then \( T_1 \) need not be included as a higher priority task when analyzing its schedulability.

The end-to-end delay bound for non-acyclic systems derived in this paper, thus enables any uniprocessor schedulability test to be used to analyze the schedulability of jobs in the distributed system. If tests such as the Liu and Layland test [12] for periodic tasks is used as the uniprocessor test, then closed-form expressions can be derived for analyzing the schedulability of tasks in distributed systems that contain cycles.

6. An Illustrative Example

In this section, we shall illustrate using a simple example, as to how the bound derived in this paper can result in tighter end-to-end delay estimates for non-acyclic task systems. We consider a system consisting of four nodes or stages, namely \( S_1, S_2, S_3, \) and \( S_4 \). We consider two tasks, \( T_1 \) and \( T_2 \), with \( T_2 \) having a higher priority than \( T_1 \). Let the period equal to the end-to-end delay of \( T_2 \) be 10 units, and that of \( T_1 \) be 12 units. Task \( T_2 \) follows the path \( S_1 - S_2 - S_3 - S_4 \), and \( T_1 \) follows the path \( S_1 - S_2 - S_3 - S_4 - S_3 - S_2 - S_1 \), as shown in Figure 4. Let the sub-job of \( T_2 \) executing on stage \( j \) be denoted as \( T_{2,j} \). The sub-jobs of \( T_1 \) are denoted as \( T_{1,1}, T_{1,2}, \ldots, T_{1,7} \) in the order in which they execute. For simplicity, let us assume that the computation times for each task on every stage is one unit. The objective is to estimate the end-to-end delay and schedulability of \( T_1 \).

Figure 4. Figure showing the paths followed by the tasks \( T_1 \) and \( T_2 \) in the example

Let us first analyze the system using holistic analysis [17]. The response time for each sub-task is at least as large as the computation time. So, the initial response times \( R_{i,j}^0 = 1 \), and the jitter for all sub-jobs is set to zero \( J_{i,j}^0 = 0 \). We now start the iterative process of estimating new response times, and updating the response times based on the jitter values. In the first iteration, each sub-job of \( T_1 \) is delayed by one invocation of \( T_2 \). Also, \( T_{1,1} \) and \( T_{1,7} \) interfere with each other as they execute on the same node (likewise, \( T_{1,2} \) and \( T_{1,6} \), \( T_{1,3} \) and \( T_{1,5} \) interfere with each other). Let us assume that a sub-job with a lower index has a higher priority. We therefore obtain \( R_{1,1}^1 = R_{1,2}^1 = R_{1,3}^1 = R_{1,4}^1 = 2 \), and \( R_{1,5}^1 = R_{1,6}^1 = R_{1,7}^1 = 3 \) (these sub-jobs are delayed by \( T_2 \) and the lower index sub-job of \( T_1 \)). We now update the jitter values as the sum of the jitter and response-time of the sub-job executing on the previous stage. That is, \( J_{1,1}^1 = 0, J_{1,2}^1 = 2, J_{1,3}^1 = 4, J_{1,4}^1 = 6, J_{1,5}^1 = 8, J_{1,6}^1 = 11, J_{1,7}^1 = 14 \). We need to follow this iterative process until convergence, but even at the first iteration the end-to-end response time of \( T_1 \) exceeds its end-to-end deadline, and \( T_1 \) is declared unschedulable. One can see that this process will quickly lead to the end-to-end response time to blow up for large systems.

Improvements to holistic analysis have been presented in [14, 15], that use the notion of offset instead of jitter. One problem with holistic analysis is that by assuming the response time at a stage to be the jitter for the next stage, the jitter values increase with longer path lengths. To overcome this problem, [14, 15] set the response time at a stage to be the offset for the next stage. The offset value denotes the minimum time after which the sub-job is activated. This makes the analysis more accurate, but more complicated as well. Using this analysis, we can obtain the response times for the sub-jobs of \( T_1 \) in the first iteration as \( R_{1,j} = 2, j = 1..7 \). Here again we need to perform an iterative process until convergence, but just
the first iteration tells us that the end-to-end response time estimate of 14 units for $T_1$ from this analysis also exceeds the end-to-end deadline of 12 units.

The fundamental problem with the above analysis is that $T_2$ delays a sub-job of $T_1$ at every stage along its path from stage $S_1$ to $S_4$ (the response time of each sub-job is calculated as 2 units). However, in reality this is not the case. When an invocation of $T_2$ delays an invocation of $T_1$ at stage $S_1$, as it has the highest priority, it will execute on future stages without waiting and hence will never delay $T_1$ on the remaining stages. By analyzing the system one stage at a time, existing analysis techniques fail to accurately account for the parallelism in the execution of different stages in the distributed system. Now, let us analyze the schedulability of $T_1$ based on the end-to-end delay bound derived in this paper. As the end-to-end deadline of $T_1$ is not larger than the period, we do not have to include $T_1$ in the set of higher priority tasks. So, $T_2$ is the only higher priority task and has only one segment with $T_1$. We therefore create a uniprocessor task $T_2^*$ with a computation time of 2 units (twice the maximum stage execution time) and period of 12 units. We construct a task $T_1^*$ with a computation time of $1 + 7 = 8$ time units (its own computation time of 1 unit and the sum of the maximum execution times of any job at each of the seven stages along the path of $T_1$). Using the response time analysis test for the hypothetical uniprocessor [1], we obtain the worst-case end-to-end response time of $T_1$ as $8 + 2 = 10$ units. Thus, $T_1$ is found to be schedulable in the original distributed system. By analyzing the system as a whole, the end-to-end delay bound derived in this paper is able to provide a more accurate bound on the end-to-end delay of tasks in distributed systems with cycles in the task graph.

7. Evaluation

In this section, we evaluate the end-to-end delay bound derived in this paper for non-acyclic systems using simulation studies for periodic tasks. We compare it with three other analysis techniques. We call the first the traditional test, that breaks the end-to-end deadline of each task into per-stage deadlines and analyzes each stage independently. If all per-stage deadlines are met then the system is deemed to be schedulable. The second test is holistic analysis applied to non-acyclic systems [17], that uses an iterative procedure to converge to worst-case response time values at each stage for every task. While extensions to holistic analysis such as [14, 15] have been proposed, these techniques also become increasingly pessimistic with system scale similar to holistic analysis, and become complex to use for large systems with hundreds of tasks. The third test is based on our own previous work for acyclic systems [8], by cutting any cycles in the system and relaxing precedence constraints (this only causes the test to be more pessimistic as an adversary has greater flexibility in choosing arrival times so as to cause worst-case delay). We do not compare with network calculus [2, 3] or its extensions such as [9], as the solution to handle cycles in the task graph requires that a system of simultaneous equations be constructed, and it may be difficult or even impossible to obtain delay bounds for certain scenarios. Further, previous comparisons such as [11] have found holistic analysis to perform better than network calculus approaches. For each test we construct an admission controller that would admit as many tasks as it can deem feasible, and measure the average per stage (resource) utilization achieved. Utilization of a stage or resource is defined as the fraction of time the resource is busy serving a task.

The schedulability test used is assumed to be deadline monotonic scheduling. We consider two types of non-acyclic traffic. The first reflects request-response type traffic, where the request follows a sequence of execution nodes, and the response follows the same set of nodes but in the opposite direction. The second traffic type emulates web server requests, where each task follows a sequence of nodes from $S_1$ to $S_n$ and returns in the opposite direction from $S_{n-1}$ to $S_1$. Thus, each task executes twice at each stage except $S_n$, once in the forward direction and once in the reverse direction. Note that in the second traffic type, each task’s path contains cycles, whereas in the first scenario, the task paths are acyclic, but with tasks going in opposite directions. The larger jitter values due to the presence of cycles in each task’s paths causes holistic analysis to perform worse in the second scenario (as observed in our simulation studies below), although the two traffic types are seemingly similar to one another.

End-to-end deadlines of tasks are chosen as $10^x a$ simulation seconds, where $x$ is a uniformly varying real value between 0 and $DR$ (deadline ratio parameter), and $a = 500 + N$, where $N$ is the number of stages on which the task executes. Such a choice of deadline values enables the deadlines of tasks to vary by a factor of $10^{DR}$. The default value of $DR$ is assumed to be 2.0. The execution times of tasks at each stage is chosen using a uniform distribution with mean $\frac{DR}{N}$, where $D$ is the end-to-end deadline, and $T$ is called the task resolution parameter. The task resolution is defined as the ratio of the sum of computation times of the task over all stages to the end-to-end deadline. The default value of $T$ is chosen as 1 : 50. The stage execution times were chosen within a range up to 10% on either side of the mean. The response time analysis technique presented in [1] is used as the schedulability test for the composed hypothetical uniprocessor for the end-to-end bound presented in this paper and the delay bound for acyclic systems in [8].

Each point in the figures below represent average values obtained from 100 executions, with each execution consisting of 80000 task invocations. For the purpose of comparing different admission controllers, each admission controller was allowed to execute on the same 100 task sets. The 95% confidence interval for all the values pre-
sent is within 1% of the mean value, and is not plotted for the sake of legibility.

Figure 5. Comparison of average per stage utilization for different number of stages in the system for request-response type traffic

In Figure 5, we compare the average per-stage utilization of the four schedulability tests for different number of nodes in the system for request-response type traffic. So, for each task there are other tasks that traverse the system in the same direction as well as in the opposite direction. The end-to-end delay bound presented in this paper is able to ensure nearly the same per-stage utilization regardless of the number of stages in the system. In contrast, all the other tests become increasingly pessimistic with system scale. The acyclic bound after cutting loops performs poorly as for each job that traverses the system in the opposite direction, the cycles are broken by cutting the job at every link creating \( N \) independent sub-jobs. These sub-jobs can therefore arrive independently of each other in a worst-case manner so as to delay the lower priority job at every stage. Holistic analysis and the traditional test analyze the system one stage at a time and fail to accurately account for the parallelism in the execution of different stages. For large systems, the jitter for downstream sub-jobs becomes large as the jitter increases with increasing number of nodes in the task path, causing holistic analysis to perform poorly for large system sizes.

For the same traffic pattern, for a system of 10 stages, we vary the deadline ratio parameter and plot the results in Figure 6. A larger value of the deadline ratio parameter implies that the range of deadline values is larger. This allows lower priority tasks with large deadlines to execute in the background of higher priority tasks with shorter deadlines, increasing the overall utilization of the system. This trend is observed for all the four schedulability tests. The new bound significantly outperforms the other tests for all deadline ratio parameter values.

Figure 6. Comparison of average per stage utilization for different deadline ratio parameter values for request-response type traffic

Figure 7. As observed in Figure 5, the new bound is able to achieve nearly the same per stage utilization regardless of system size. Also note that holistic analysis and the traditional test perform poorly for this traffic scenario compared to their achieved utilization under request-response type traffic shown in Figure 5. For holistic analysis, the jitter values increase considerably due to the presence of cycles in the task path and the large path length causing the analysis to be extremely pessimistic. The traditional test breaks the end-to-end deadline into per-stage deadlines, which works poorly when the path length is long, as the delay experienced by tasks at different stages is not uniform.

Figure 8, presents a comparison of the four schedulability tests for different deadline ratio parameter values for the web server type traffic scenario in a system with 10 stages. As observed in Figure 7, holistic analysis and the traditional test perform poorly for this traffic scenario. The utilization values are observed to increase with increasing deadline ratio parameter values, as low priority jobs with large deadlines are able to execute in the background of higher priority jobs with short deadlines, thereby increas-
In this paper, we derive an end-to-end delay bound for tasks in distributed systems with cycles in the task graph. The bound allows one of several schedulability tests to be performed, enabling the construction of the first generalized closed form expression for analyzing the schedulability of non-acyclic flows in distributed systems. This is in contrast to existing analysis techniques that involve iterative solutions or provide no solutions at all when tasks are non-acyclic. We show using simulation studies that the schedulability tests constructed in this paper are less pessimistic compared to existing analysis techniques for such systems with long path lengths.

8. Conclusion

In this paper, we derive an end-to-end delay bound for tasks in distributed systems with cycles in the task graph. The bound allows one of several schedulability tests to be performed, enabling the construction of the first generalized closed form expression for analyzing the schedulability of non-acyclic flows in distributed systems. This is in contrast to existing analysis techniques that involve iterative solutions or provide no solutions at all when tasks are non-acyclic. We show using simulation studies that the schedulability tests constructed in this paper are less pessimistic compared to existing analysis techniques for large systems. We envision that this result will foster more research and a better understanding of timing behavior in distributed systems.

References