Real-Time Syst (2010) 46: 121–151
DOI 10.1007/s11241-010-9098-1

Reduction-based schedulability analysis of distributed systems with cycles in the task graph

Praveen Jayachandran · Tarek Abdelzaher

Published online: 30 June 2010

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. Using the end-to-end delay bound, we extend the delay composition algebra developed for acyclic distributed systems in prior work, to handle loops in the task graph as well. Evaluation shows that the schedulability tests thus constructed are less pessimistic than prior approaches for large distributed systems.

Keywords Schedulability analysis · End-to-end delay · Real-time distributed system · Non-acyclic systems · Problem reduction

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,

This work was funded in part by NSF grants CNS 05-53420, CNS 06-13665, and CNS 07-20513, and ONR grant N00014-10-1-0172.

P. Jayachandran (✉) · T. Abdelzaher
Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
e-mail: pjayach2@uiuc.edu

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 (Jayachandran and Abdelzaher 2007) and distributed systems of directed acyclic task graphs (DAGs) (Jayachandran and Abdelzaher 2008c). An algebra was described for *reducing* such DAGs to equivalent uniprocessors (Jayachandran and Abdelzaher 2008a) 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 (Cruz 1991a, 1991b) and holistic analysis (Tindell and Clark 1994), together with various extensions such as Pellizzoni and Lipari (2005). 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 does contain cycles, this technique breaks down due to the cyclic dependency. To overcome this problem, an iterative procedure is suggested in prior literature (Palencia and Harbour 2003; Pellizzoni and Lipari 2005) 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 job-level dynamic priority scheduling (e.g. EDF), 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 (Jayachandran and Abdelzaher 2008c), 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. A different hypothetical uniprocessor is created for the schedulability analysis of each task in the distributed system. Further, we use the new end-to-end delay bound to generalize the Delay Composition Algebra presented in Jayachandran and Abdelzaher (2008a) to handle non-acyclic systems, by defining a new operator called `LOOP`. The `LOOP` operator permits composition of resource nodes when tasks traverse them in different orders, or when tasks revisit a particular node. The algebra enables us to compose together workload in different nodes, eventually reducing any arbitrary distributed system to a single equivalent uniprocessor workload. The equivalent uniprocessor workload can then be analyzed to infer the end-to-end delay and schedulability properties of all the tasks in the original non-acyclic distributed system. Thus, the algebra allows the reduction for analyzing the schedulability of all the tasks in the system to be performed simultaneously.

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 uniprocessors, 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 Sect. 2. We describe the system model in Sect. 3 and state and prove the end-to-end delay bound for jobs in non-acyclic systems in Sect. 4. In Sect. 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 Sect. 6. In Sect. 7, we present simulation studies to evaluate the delay bound. We present the extension of the Delay Composition Algebra to handle non-acyclic systems in Sect. 8. We illustrate the algebra with an example in Sect. 9. We conclude the paper in Sect. 10. In Appendix, we present the proof of correctness of the algebra.

2 Related work

The problem of analyzing the schedulability of tasks in distributed systems has been studied using various techniques in the past. However, these techniques become more pessimistic or provide no solutions when the task graph contains cycles and when the system 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 Xu and Parnas (1993), Fohler and Ramamritham (1997) 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, Kao and Garcia-Molina (1997), Zhang et al. (2005) 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 Natale and Stankovic

(1994). 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 Tindell and Clark (1994), and has since had several extensions such as Palencia and Harbour (2003), Pellizzoni and Lipari (2005) 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 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 defines 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 Palencia and Harbour (2003), Pellizzoni and Lipari (2005) 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 (Cruz 1991a, 1991b) 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 Thiele et al. (2000), and has since been extended to handle different system models such as Jonsson et al. (2008), Wandeler et al. (2004). 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 Cruz (1991b), 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. 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 Koubaa and Song (2004), where holistic analysis was found to be less pessimistic than network calculus in general. We show in the evaluation section that 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 Jayachandran and Abdelzaher (2007, 2008b). This was extended to DAGs, and also to partitioned resources (e.g., TDMA scheduling) in Jayachandran and Abdelzaher (2008c). An algebra called Delay Composition Algebra was developed to reduce

acyclic distributed systems to equivalent uniprocessors for the purpose of schedulability analysis in Jayachandran and Abdelzaher (2008a). In this paper, we provide an end-to-end delay bound for tasks in distributed systems that contain cycles in the task graph. Further, we show how the algebra presented in Jayachandran and Abdelzaher (2008a), can be extended to handle non-acyclic systems. This is achieved by defining a new operator in the algebra called the LOOP.

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. This is not too restrictive a limitation, as this is a feasible assumption for several classes of systems including automobile systems, performance-sensitive server farms, and avionics. 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 (Jayachandran and Abdelzaher 2007, 2008a, 2008c), 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 xth fold of job J_k by J_k^x. 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_k^1 and J_k^2 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 Fig. 1. The task graph for this system contains a loop, as shown in Fig. 1, even though individual jobs do not.

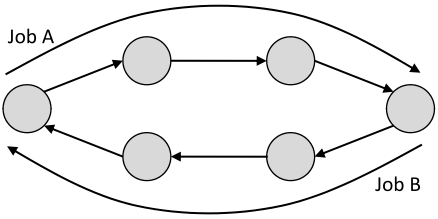

Fig. 1 A task graph with a cycle

Hence, loops in the task graph capture cyclic dependencies that may involve one or more jobs.

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_k 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-acyclic 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 under preemptive scheduling. 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. Towards the end of this section, we state the end-to-end delay bound when the scheduling is non-preemptive, and omit the proof in the interest of brevity.

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 denote the xth 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

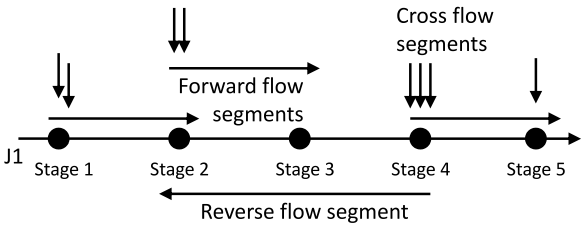

Fig. 2 Three segment types

(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_i^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 $J_i^{x,s}$ 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.

Consider the interval of time starting from the arrival time of J_1 to the system, until the finish time of J_1 on its 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:

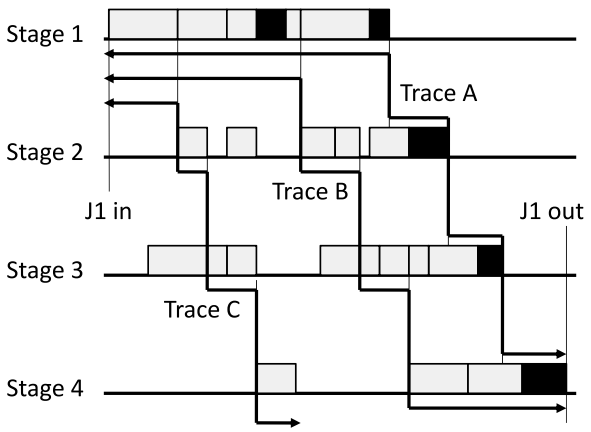

Fig. 3 An execution trace

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 Fig. 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 Fig. 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 $Node_{j,max}$ to denote the maximum stage execution time of all job-segments $J_i^{x,s}$ on node j. The three lemmas below bound delays due to the three types of segments depicted in Fig. 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 $C_{i,max}^{x,s}$).*

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 $C_{i,max}^{x,s}$).*

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 $J_i^{x,s}$ 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*:

$$\sum_{\text{segments}} C_{i,max}^{x,s} + \sum_{\substack{\text{forward-flow}\\ \text{segments}}} C_{i,max}^{x,s} + \sum_{j \in p_1} Node_{j,max} \tag{1}$$

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 that preempt other segments and execute ahead of lower priority segments that arrived earlier at the stage, denoted by C_{ff_2}, and (iii) the total length of stage execution times of segments not at their end stages, and that do not preempt another segment, 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 $C_{i,max}^{x,s}$. The total of all end-stage computation times over all segments is thus given by:

$$C_{ff_1} \leq \sum_{\substack{\text{forward-flow} \\ \text{segments}}} C_{i,max}^{x,s} \tag{2}$$

To bound, C_{ff_2}, note that, each segment can preempt another segment at most once along the earliest traversal trace. Consider a segment $J_i^{x,s}$ that preempts another segment in the earliest traversal trace at stage j. By definition of the earliest traversal trace (see Definition 2), starting from the time this preemption occurs, no segment of lower priority than $J_i^{x,s}$ can be a part of the earliest traversal trace until the end stage of $J_i^{x,s}$. Therefore, $J_i^{x,s}$ will not preempt any other segment in the earliest traversal trace. Thus, the total length of stage execution times of segments in the earliest traversal trace that preempt and execute ahead of lower priority segments that arrived earlier is bounded by:

$$C_{ff_2} \leq \sum_{\text{segments}} C_{i,max}^{x,s} \tag{3}$$

To bound C_{ff_3}, observe that, there exists at most one execution time of a segment at each stage of the earliest traversal trace that is not an end stage of a segment and that does not execute ahead of a lower priority segment that arrived earlier in the earliest traversal trace (that is, not bounded by C_{ff_1} or C_{ff_2}). Let us assume the contrary and suppose that there are two execution times of segments J_i and J_k at a stage j in the earliest traversal trace that are not included in C_{ff_1} or C_{ff_2}. Without loss of generality, let us also assume that J_i arrives at stage j before J_k. Now, J_k cannot be a higher priority segment that arrives after J_i and completes execution before J_i (covered under C_{ff_2}). Thus, J_k can start executing on stage j only after J_i completes execution. As stage j is not the end stage of J_i, by definition, the portion of the earliest traversal trace on stage j should end with the execution of J_i and cannot include the execution of J_k, resulting in a contradiction. Therefore, there exists at most one execution time of a segment at each stage of the earliest traversal trace that is not bounded under C_{ff_1} or C_{ff_2}. Thus,

$$C_{ff_3} \leq \sum_{j \in p_1} Node_{j,max} \tag{4}$$

Adding up C_{ff_1}, C_{ff_2} and C_{ff_3}, given by (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 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*:

$$Delay(J_1) \leq \sum_{i} \sum_{J_i^x \in Q_i} \sum_{J_i^{x,s} \in Seg_i^x} 2C_{i,max}^{x,s} + \sum_{j \in p_1} Node_{j,max} \tag{5}$$

Proof The theorem follows trivially from Lemmas 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. Let $Node_{j,all\_max}$ denote the maximum computation time of any job (not just higher priority jobs) on stage j, and let $Node_{j,lower\_max}$ 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*:

$$Delay(J_1) \leq \sum_{i} \sum_{J_i^s \in Q_i} \sum_{J_i^{x,s} \in Seg_i^x} C_{i,max}^{x,s} + \sum_{j \in p_1} Node_{j,all\_max} + \sum_{j \in p_1} Node_{j,lower\_max} \tag{6}$$

The above delay bound for any job can be calculated in $O(MN)$ time, where N is the number of stages in the system and M is the number of tasks. Each higher priority task's path can be broken down into various segments and the maximum computation time for the task on each of its segments can be calculated in $O(N)$ time. This has to be repeated for at most M tasks. Likewise, the maximum computation time of higher priority tasks on a stage, $Node_{j,max}$, can be calculated in $O(M)$ time and this needs to be repeated for at most N stages. Therefore, the net complexity of calculating the delay bound is $O(MN)$. In contrast, existing techniques to calculate the end-to-end delay bound for tasks such as holistic analysis and network calculus, have a pseudo-polynomial time complexity as they involve an iterative solution until convergence is reached.

5 Schedulability analysis

In the previous section, we derived the end-to-end delay bound assuming knowledge of all higher priority jobs that are concurrently present in the system. We were not concerned with how to determine such a set of higher priority jobs that interfere with the job under consideration (for instance, determining the number of instances of a higher priority periodic task that interfere with the job under consideration). It simply proves a fundamental property of delay composition over any such set. However, for

the purpose of schedulability analysis, it is important to determine this set of higher priority jobs. Trivially, in the worst case, this set will include all jobs whose active intervals (interval between their arrival time to the system and their deadline) overlap that of the job under consideration. By reducing the problem of schedulability analysis to that of analyzing an equivalent uniprocessor, similar to the technique presented in Jayachandran and Abdelzaher (2008c), we delegate the problem of identifying the set of interfering higher priority jobs to the uniprocessor schedulability analysis used.

To analyze the schedulability of a job J_1, the transformation is carried forth as follows:

- Each higher priority job-segment $J_i^{x,s}$ in the distributed system, is replaced by a uniprocessor job $J_i^{x,s*}$ with computation time equal to $2C_{i,max}^{x,s}$ and same deadline as J_i;
- Job J_1 is replaced by a uniprocessor job J_1^* with computation time equal to $C_{1,max} + \sum_{j \in p_1} Node_{j,max}$ and deadline same as J_1

If the uniprocessor job J_1^* is schedulable, so is job J_1 in the original distributed system (proof of this statement is similar to the description in Jayachandran and Abdelzaher 2008c). The uniprocessor is constructed such that the worst-case delay of J_1^* in the uniprocessor is no less than the end-to-end delay bound of J_1 in the distributed system, as derived in the previous section. If the delay of J_1^* on the uniprocessor is less than its deadline (inferred using uniprocessor schedulability analysis), the delay of J_1 in the distributed system will also be less than its deadline. 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 $\lceil D_i/P_i \rceil$. Further, if the task T_1 being analyzed has cycles in its path, then earlier invocations of T_1 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. A more detailed description of how the reduction can be performed for periodic tasks is provided in our earlier work (Jayachandran and Abdelzaher 2008b).

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 (Liu and Layland 1973) 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,

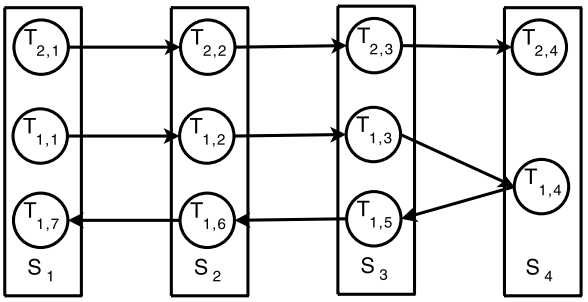

Fig. 4 Figure showing the paths followed by the tasks T_1 and T_2 in the example

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 deadline 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 Fig. 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.

Let us first analyze the system using holistic analysis (Tindell and Clark 1994). The response time for each sub-task is at least as large as the computation time. So, the initial response times $R^0_{1,j} = 1$, and the jitter for all sub-jobs is set to zero $J^0_{1,j} = 0$. 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 $R^1_{1,1} = R^1_{1,2} = R^1_{1,3} = R^1_{1,4} = 2$, and $R^1_{1,5} = R^1_{1,6} = R^1_{1,7} = 3$ (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, $J^1_{1,1} = 0$, $J^1_{1,2} = 2$, $J^1_{1,3} = 4$, $J^1_{1,4} = 6$, $J^1_{1,5} = 8$, $J^1_{1,6} = 11$, $J^1_{1,7} = 14$. 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 Palencia and Harbour (2003), Pellizzoni and Lipari (2005), 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 (Palencia and Harbour 2003; Pellizzoni and Lipari 2005) 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 $R_{1,j} = 2$, for $j = 1..7$. 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 proposed by Audsley et al. (1993) for the hypothetical uniprocessor, 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 (Tindell and Clark 1994), 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 Palencia and Harbour (2003), Pellizzoni and Lipari (2005) 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 (Jayachandran and Abdelzaher 2008c), 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 (Cruz 1991a, 1991b) or its extensions such as Jonsson et al. (2008), 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 Koubaa

and Song (2004) 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 default value of the total number of tasks presented to the admission controllers was increased as a function of system size, maintaining the ratio of tasks to stages to be 25.

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 cyclic 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. In the first scenario, the request and the response are separate tasks, but in the second scenario, a single task includes the forward and reverse paths. 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 $a = 500 \cdot N$, 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. Task periods are assumed to be equal to the end-to-end deadline. The execution times of tasks at each stage is chosen using a uniform distribution with mean $\frac{DT}{N}$, 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 by Audsley et al. (1993) 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 Jayachandran and Abdelzaher (2008c).

Each point in the figures below represent average values obtained from 100 executions, with each execution consisting of 80000 task invocations of all tasks in the system taken together. 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 presented is within 1% of the mean value, and is not plotted for the sake of legibility.

In Fig. 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

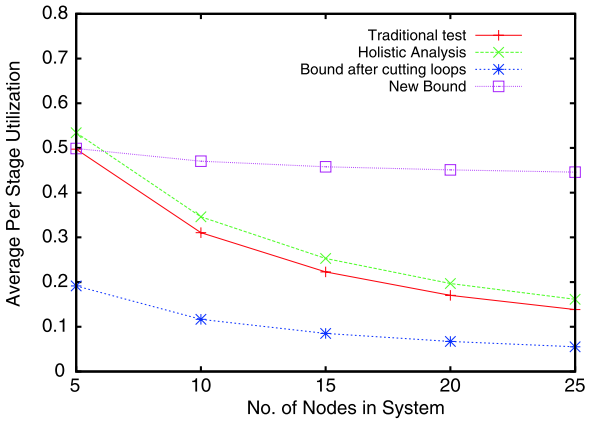

Fig. 5 Comparison of average per stage utilization for different number of stages in the system for request-response type traffic

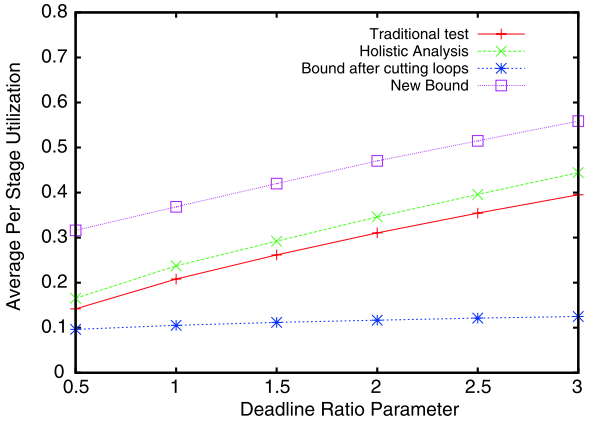

Fig. 6 Comparison of average per stage utilization for different deadline ratio parameter values for request-response type traffic

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 Fig. 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

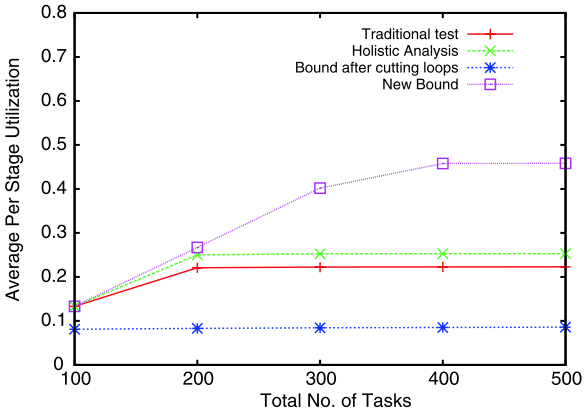

Fig. 7 Comparison of average per stage utilization for different values of the number of tasks presented to the admission controllers

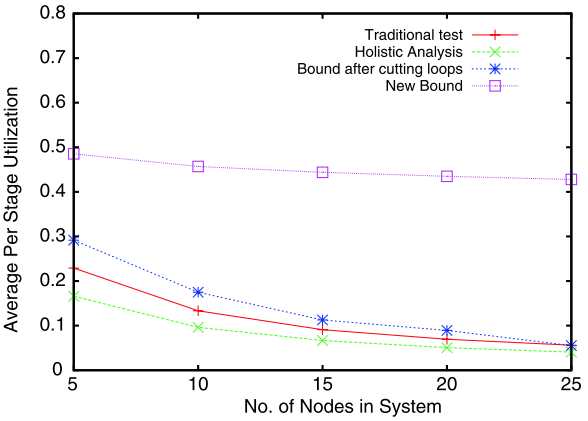

Fig. 8 Comparison of average per stage utilization for different number of stages in the system for the cyclic requests traffic type

tests for all deadline ratio parameter values and experiences a similar (seemingly linear) improvement with increasing DR values as the holistic and traditional tests.

For a system with 15 stages and a deadline ratio parameter of 2, we vary the number of tasks presented to the admission controllers. We plot the average per stage utilization as a function of the number of tasks in Fig. 7. For a small system load of 100 tasks, we notice that all the schedulability tests admit all the tasks, except the test that cuts cycles and relaxes precedence constraints. As the offered load increases, we notice that our new analysis technique is able to admit more tasks than the other schedulability tests and achieve a higher average per-stage utilization. There is no increase in the average per-stage utilization beyond 400 tasks, as no more tasks can be admitted. This demonstrates that the test performs well regardless of the load offered to the system.

For the cyclic requests traffic type, where each task traverses the stages in the system in the forward direction and then in the reverse direction, we plot the average per stage utilization for increasing number of stages in the system in Fig. 8. As observed in Fig. 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

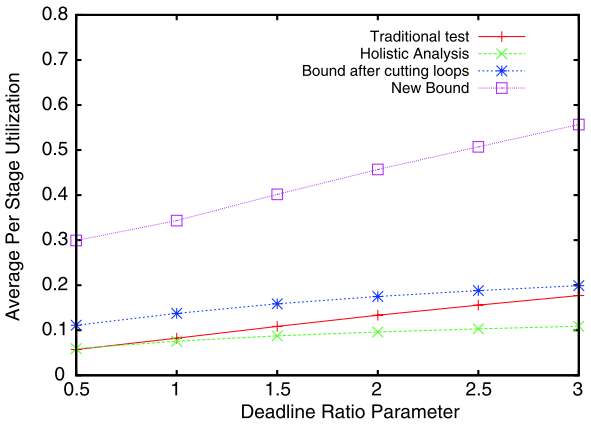

Fig. 9 Comparison of average per stage utilization for different deadline ratio parameter values for the cyclic requests traffic type

perform poorly for this traffic scenario compared to their achieved utilization under request-response type traffic shown in Fig. 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 9, presents a comparison of the four schedulability tests for different deadline ratio parameter values for the cyclic requests traffic type scenario in a system with 10 stages. As observed in Fig. 8, 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 increasing the overall utilization of each stage. The new bound significantly outperforms the other schedulability tests for such systems with long path lengths.

8 Delay composition algebra: handling loops

In Jayachandran and Abdelzaher (2008a), we developed *delay composition algebra* for acyclic distributed systems. The operands of the algebra represented workload of individual resource nodes or composed sub-systems. By systematically applying a set of operators on the operands, the algebra enabled the reduction of an acyclic distributed system workload into an equivalent hypothetical uniprocessor workload for the purposes of schedulability analysis. The algebra enables the reduction process in the schedulability analysis of all tasks in the system to be performed concurrently. Using the delay composition theorem derived in this paper, we can generalize the delay composition algebra to handle loops in the task graph. This is done by introducing a new operator called the LOOP and modifying the definition of the SPLIT operator. Also, the representation of the workload at each resource stage needs to be augmented to account for the fact that a job may visit a stage multiple times. We describe our operand representation in Sect. 8.1. Next we define operators in Sect. 8.2,

to systematically compose together workloads on resource nodes in the task graph, until only a single node remains. The workload on this node represents a uniprocessor job set, which can be studied using any traditional uniprocessor analysis to infer the schedulability of jobs in the original distributed system. For a proof that the algebra indeed computes the worst-case delay as per the delay composition theorem, readers are referred to Appendix.

8.1 Operand representation

The operand representation is similar to the representation in Jayachandran and Abdelzaher (2008a). The operand is an $n \times n$ array of delay terms, with the $(i,k)^{\text{th}}$ element denoting the delay that job J_i causes job J_k. As motivated in Jayachandran and Abdelzaher (2008a), the delay that each job J_i causes another job J_k is maintained as two terms—a max-term and an accumulator-term. Accordingly, each element (i,k) is a two-tuple $(q_{i,k}, r_{i,k})$, where the first term $q_{i,k}$ denotes the max-term and the second term $r_{i,k}$ denotes the accumulator-term. The max-term $q_{i,k}$ denotes the worst-case interference that the current segment of J_i (the segment that includes the node under consideration) causes J_k. The accumulator-term $r_{i,k}$ denotes the worst-case interference due to all previous segments of J_i (segments earlier in the path of J_i than the current segment). The matrix has an additional row, in which the k^{th} element s_k represents the stage-additive component of the delay of job J_k that is independent of the number of jobs in the system and is only dependent on the number of stages on which J_k executes.

The matrix for a single stage j is constructed similar to the description in Jayachandran and Abdelzaher (2008a). Without loss of generality, let jobs be indexed in order of priority and $i < k$ imply that J_i has a higher priority than J_k. Consider a job J_k and the column corresponding to it. The accumulator term $r_{i,k}$ is set to zero, for all i. If J_k does not execute at stage j, then $q_{i,k}$ and s_k are set to zero, for all i. If J_k executes at stage j, but a job J_i does not or if it has a lower priority than J_k, then $q_{i,k}$ is set to zero. If J_i executes on stage j exactly once, then $q_{i,k}$ is set to $C_{i,j}$. If J_i visits stage j multiple times, then $q_{i,k}$ is set to the maximum computation time of J_i over all its visits to stage j. The stage-additive component, s_k is defined as the maximum computation time of any higher priority job on stage j, counted as many times as J_k visits the stage. Suppose that J_k visits the stage p times, then $s_k = p \times \max_{i \leq k} C_{i,j}$.

Under non-preemptive scheduling, the matrix is constructed in a very similar manner, except for the stage-additive component s_k, which is defined as the sum of two terms. The first term is the maximum computation time of any job (not just higher priority jobs) on stage j, and the second term is the maximum computation time of any lower priority job on stage j, each counted p times. That is, $s_k = p(\max_i C_{i,j} + \max_{i>k} C_{i,j})$.

8.2 Definition of operators

The algebra consists of three main operators, namely PIPE, SPLIT, and LOOP. The operators ensure that in the resultant operand matrix, every term $(q_{i,k}, r_{i,k})$ accurately represents the max-term and accumulator-term of the worst-case delay that job J_i causes J_k over the set of stages the operand represents.

8.2.1 The PIPE operator

The PIPE operator can be applied to two nodes to compose them into a single node, whenever the upstream node has exactly one outgoing arc. The PIPE operator is defined as follows:

Definition (PIPE operator) For *any* two neighboring nodes in the resource graph, represented by operand matrices A and B, if the upstream node has exactly one outgoing arc, the two nodes can be composed into a single node represented by matrix C using the PIPE operator. $C = A\ PIPE\ B$ is obtained as follows:

1. $\forall i, k$: $q^C_{i,k} = \max(q^A_{i,k}, q^B_{i,k})$
2. $\forall i, k$: $r^C_{i,k} = r^A_{i,k} + r^B_{i,k}$
3. $\forall k$: $s^C_k = s^A_k + s^B_k$

8.2.2 The SPLIT operator

The SPLIT operator is used on a node that has more than one outgoing arc to break it into multiple nodes one for each outgoing arc. The definition of the SPLIT operator is similar to the description in Jayachandran and Abdelzaher (2008a), except that we relax the precondition that the node being split, say node j, should have no incoming arcs. We replace this precondition with the condition that an individual outgoing arc l can be split (creating a separate node) as long as all the jobs traversing the arc in question have node j as their start node (they should not be traversing any incoming arc of node j). Outgoing arcs from node j that do not satisfy this condition cannot be split. The load matrix A of node j is split into two matrices, one for node j and one for the new node j' that is created. The resultant matrix for the new node j' is obtained by replicating matrix A and zeroing out all columns corresponding to jobs that do not traverse arc l, and the matrix for node j is obtained by zeroing out all columns corresponding to jobs that traverse arc l. Further, for any job J_k and a higher priority job J_i, if the two jobs follow different outgoing arcs from node j, the accumulator term of J_k (in the output matrix containing J_k) is updated by replacing the element $(q_{i,k}, r_{i,k})$ with $(0, q_{i,k} + r_{i,k})$.

Definition (SPLIT operator) Let matrix A denote node j and let l be an outgoing arc of node j, such that all jobs X_1 traversing arc l have node j as their start stage. Let X_2 denote the set of jobs that do not traverse arc l. The resultant matrices A_x, $x = 1, 2$, are obtained as follows:
$\forall J_k$:

1. if $J_k \in X_x$:
   $s^{A_x}_k = s^A_k$; $\forall i$: if $J_i \in X_x$: $q^{A_x}_{i,k} = q^A_{i,k}$, $r^{A_x}_{i,k} = r^A_{i,k}$, else $q^{A_x}_{i,k} = 0$, $r^{A_x}_{i,k} = q^A_{i,k} + r^A_{i,k}$.
2. if $J_k \notin X_x$:
   $s^{A_x}_k = 0$; $\forall i$: $q^{A_x}_{i,k} = 0$, $r^{A_x}_{i,k} = 0$.

8.2.3 The LOOP operator

Any situation where a PIPE or SPLIT operation cannot be applied to any arc in the graph, implies that a loop exists in the task graph (for a proof of this statement, refer to the proof of liveness in Jayachandran and Abdelzaher (2008a)). Consider an outgoing arc l from a node j that is part of a loop. Let X denote the set of jobs that traverse arc l. If the set of jobs that traverse the other outgoing arcs from node j is a subset of X, then the LOOP operator can be applied to arc l. This condition ensures that there is no job whose path is splitting away from the jobs traversing arc l on which the LOOP operator is applied. Like the PIPE operator, the LOOP composes the two nodes A and B at the ends of link l together into a single node. It takes the maximum of corresponding max-terms and the sum of the corresponding accumulator terms in the two operand matrices. If composing the two nodes marks the end of a higher priority task segment, say for task i (all other arcs in the segment have been composed together), then the corresponding resultant max-term $q_{i,k}$ is added to the accumulator $r_{i,k}$, and the max-term is reset to the maximum computation time of J_i on the stage (A or B) from which its next segment starts. Further, if the higher priority task J_i traverses both the forward and the reverse link (that is, traverses the link from A to B as well as the link from B to A), then we add twice the resultant max-term to the accumulator term to account for the interference due to both the forward and reverse flow segments. If a loop is traversed by two tasks J_i and J_k p times (the same sequence of links), then we account for p times the delay component to be added to the accumulator term.

Definition (LOOP operator) When a PIPE or a SPLIT operation cannot be performed, and node j has an outgoing arc l that is part of a loop, such that the set of all jobs that traverse other outgoing arcs from node j is a subset of the set of jobs that traverse the outgoing arc l from node j, then the LOOP operator can be applied to arc l. Let A and B represent the operand matrices of the nodes that arc l connects, and let C be the resultant operand matrix. $C = A\ LOOP\ B$, is obtained as follows:

1. $\forall i,k$: $q^C_{i,k} = \max(q^A_{i,k}, q^B_{i,k})$; $r^C_{i,k} = r^A_{i,k} + r^B_{i,k}$
2. $\forall i,k$: If end of higher priority segment of J_i (J_i and J_k traverse loop p times):
   - 2.1 If J_i traverses both the arc from A to B as well as the arc from B to A, then $r^C_{i,k} = r^C_{i,k} + 2p \times q^C_{i,k}$
     else $r^C_{i,k} = r^C_{i,k} + p \times q^C_{i,k}$
   - 2.2 If J_i has outgoing arc from node corresponding to A, then $q^C_{i,k} = q^A_{i,k}$
     else if J_i has outgoing arc from node corresponding to B, then $q^C_{i,k} = q^B_{i,k}$
     else $q^C_{i,k} = 0$
3. $\forall k$: $s^C_k = s^A_k + s^B_k$

8.2.4 The CUT operator

When the LOOP operator cannot be performed as well, then the CUT operator as defined in Jayachandran and Abdelzaher (2008a) needs to be performed to break a loop in the task graph. Such a situation might arise as the LOOP operator can only

be applied to a link l if the set of jobs traversing link l is a superset of the set of jobs traversing other outgoing arcs from the node at the head of link l.

The CUT operation breaks each job traversing the arc being cut into two independent jobs, one for the part before the cut and one for the part after. As explained in Jayachandran and Abdelzaher (2008a), this operation only relaxes constraints on the arrival times of jobs, allowing jobs to arrive in a manner that can cause worst-case delay (an adversary has greater freedom in choosing the arrival times of jobs to cause a worst-case delay). This decreases the schedulability of the task set and performs a transformation that is safe.

Definition (CUT operator) When the directed resource graph contains a cycle and when a PIPE, SPLIT, or LOOP operation cannot be performed, a CUT operation can be performed on one of the arcs forming the cycle. Each job crossing that arc is thereby replaced by two independent jobs; one for the part before the cut and one for the part remaining. Each new job will have a separate row and column in the operand matrices for stages on which they execute.

The definition of the operators are the same regardless of whether the scheduling is preemptive or non-preemptive. By successively applying the operators of the algebra, the distributed system can be reduced to a single equivalent uniprocessor. Note that as the max and sum operations are commutative and associative, the PIPE and LOOP operators are commutative and associative as well.

The proof of liveness, that is to show that repeatedly applying the operators of the algebra always reduces the graph to a single node, follows from the liveness property of the PIPE and SPLIT operators proved in Jayachandran and Abdelzaher (2008a). When the graph has loops, then either the LOOP operator can be applied, or the CUT operator can be applied to break the loop.

8.3 Analyzing the equivalent uniprocessor

Upon reducing the distributed system to a single node, the end-to-end delay and schedulability of a job J_k can be inferred from the k^{th} column in the load matrix. For each i, k, the term $(q_{i,k}, r_{i,k})$ is translated into $(0, q_{i,k} + r_{i,k})$. The transformation is similar to the description in Jayachandran and Abdelzaher (2008a), and is carried forth as follows:

- Each task T_i, $i \neq k$ in the original distributed system is transformed to task T_i^* on a uniprocessor, with a computation time $C_i^* = r_{i,k}$, if scheduling is non-preemptive, or $C_i^* = 2r_{i,k}$, if scheduling is preemptive (the reason for which is explained in the proof in Appendix). The period P_i (if J_i is periodic) or minimum inter-arrival time (if it is sporadic) remains the same (i.e., $P_i^* = P_i$).
- Task T_k, for which schedulability analysis is performed, is transformed to task T_k^* with $C_k^* = r_{k,k}$ plus an extra task of computation time s_k. The period or minimum inter-arrival time for both, remains that of T_k.

We prove in Appendix that if T_k^* meets its deadline on the uniprocessor when scheduled together with this hypothetical task set, then T_k meets its deadline in the original distributed system. Any uniprocessor schedulability test can be used to analyze the schedulability of T_k^*. Note that a separate test is needed per task.

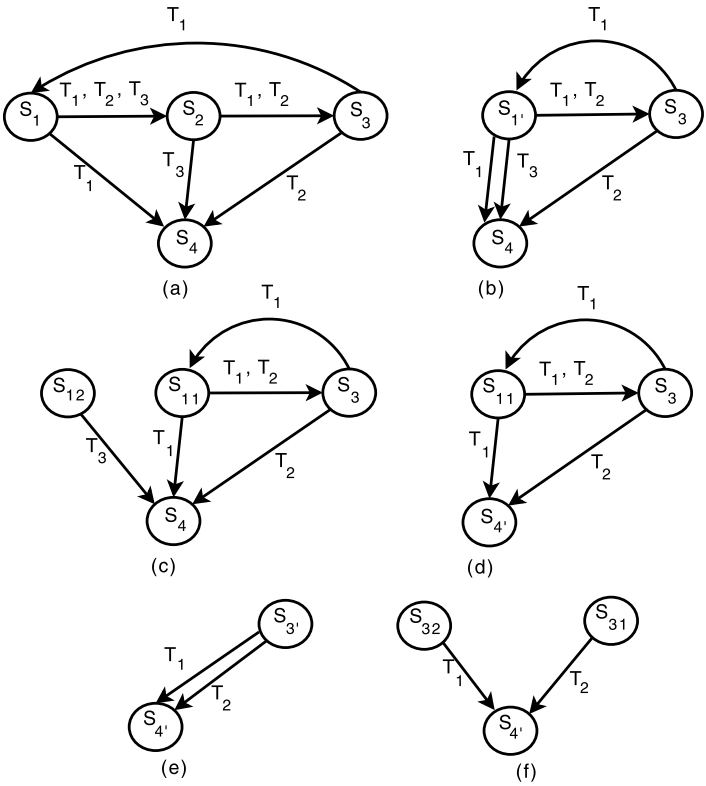

Fig. 10 (**a**) Example system to be composed. (**b**) Composed system after step 1. (**c**) Composed system after step 2. (**d**) After step 3. (**e**) After step 4. (**f**) After step 5

9 Example illustrating the algebra

We now illustrate how the algebra can be applied to a distributed system to reduce it to a single equivalent hypothetical uniprocessor for the purpose of analyzing the end-to-end delay and schedulability of jobs in the original distributed system. We consider a system of four resource stages shown in Fig. 10(a), and three periodic tasks T_1, T_2, and T_3, in decreasing priority order. T_1 follows the path S_1–S_2–S_3–S_1–S_4, T_2 follows S_1–S_2–S_3–S_4, and T_3 follows S_1–S_2–S_4. Each task invocation requires one unit of computation time at each resource along its path, and the relative end-to-end deadline is assumed to be the same as the task period. T_1 has a period of 10 units, and T_2 and T_3 have a period of 20 units. We do not need to create a virtual finish node as all task routes end at the same finish node (S_4).

Let A_i denote the operand matrix for stage S_i. The initial operand matrices are constructed as shown below. As task T_1 executes twice on stage S_1, the stage-additive

component s_1 of A_1 is two, while all other stage-additive component values are one.

$$A_1 = \left(\begin{array}{c|ccc} & T_1 & T_2 & T_3 \\ \hline T_1 & (1,0) & (1,0) & (1,0) \\ T_2 & (0,0) & (1,0) & (1,0) \\ T_3 & (0,0) & (0,0) & (1,0) \\ & \cdots\cdots\cdots\cdots\cdots & & \\ & 2 & 1 & 1 \end{array}\right)$$

$$A_2 = A_4 = \left(\begin{array}{c|ccc} & T_1 & T_2 & T_3 \\ \hline T_1 & (1,0) & (1,0) & (1,0) \\ T_2 & (0,0) & (1,0) & (1,0) \\ T_3 & (0,0) & (0,0) & (1,0) \\ & \cdots\cdots\cdots\cdots\cdots & & \\ & 1 & 1 & 1 \end{array}\right)$$

$$A_3 = \left(\begin{array}{c|cc} & T_1 & T_2 \\ \hline T_1 & (1,0) & (1,0) \\ T_2 & (0,0) & (1,0) \\ & \cdots\cdots\cdots\cdots & \\ & 1 & 1 \end{array}\right)$$

All nodes have 2 out-going arcs, and no PIPE or SPLIT operations can be performed. A loop exists, and we apply the LOOP operator to arc S_1–S_2.

Step 1: A_1 *LOOP* $A_2 = A_{1'}$

We take the maximum of the corresponding max-terms and the sum of the corresponding accumulator terms and the stage-additive components. The arc under consideration does not mark the end of T_1's segment when considering the delay of T_2. But, it marks the end of the segment of T_1 that interferes with T_3. As T_3 executes on only one of the arcs and does not traverse an arc from S_2 to S_1, it contributes only one unit of delay, which is added to the accumulator term. We obtain $A_{1'}$ as,

$$A_{1'} = \left(\begin{array}{c|ccc} & T_1 & T_2 & T_3 \\ \hline T_1 & (1,0) & (1,0) & (1,1) \\ T_2 & (0,0) & (1,0) & (1,0) \\ T_3 & (0,0) & (0,0) & (1,0) \\ & \cdots\cdots\cdots\cdots\cdots & & \\ & 3 & 2 & 2 \end{array}\right)$$

The resultant task graph is as shown in Fig. 10(b). Now, stage $S_{1'}$ is the start stage for T_3, and T_3 is the only job that traverses the arc from $S_{1'}$ to S_4 (note that T_1 traverses a different arc from $S_{1'}$ to S_4). We can therefore apply the SPLIT operator to split T_3 along that arc creating nodes S_{11} and S_{12}, whose operand matrices are as follows:

Step 2: $SPLIT(S_{1'}, \{T_3\}) \Rightarrow A_{11}, A_{12}$

$$A_{11} = \left(\begin{array}{c|cc} & T_1 & T_2 \\ \hline T_1 & (1,0) & (1,0) \\ T_2 & (0,0) & (1,0) \\ T_3 & (0,0) & (0,0) \\ & \cdots\cdots\cdots\cdots & \\ & 3 & 2 \end{array}\right), \qquad A_{12} = \left(\begin{array}{c|c} & T_3 \\ \hline T_1 & (0,2) \\ T_2 & (0,1) \\ T_3 & (1,0) \\ & \cdots\cdots \\ & 2 \end{array}\right)$$

Figure 10(c) shows the updated task graph. S_{12} can now be piped with S_4 to give $S_{4'}$.

Step 3: $A_{12}\ PIPE\ A_4 = A_{4'}$

$$A_{4'} = \left(\begin{array}{c|ccc} & T_1 & T_2 & T_3 \\ \hline T_1 & (1,0) & (1,0) & (1,2) \\ T_2 & (0,0) & (1,0) & (1,1) \\ T_3 & (0,0) & (0,0) & (1,0) \\ & \cdots\cdots\cdots\cdots\cdots\cdots & & \\ & 1 & 1 & 3 \end{array}\right)$$

The resultant task graph is shown in Fig. 10(d). Again nodes have more than one out-going arc and no PIPE or SPLIT operations can be performed. We perform a LOOP operation on the arc S_{11}–S_3 to merge the nodes into a single node $S_{3'}$. This operation marks the end of the task segment of T_1 that delays T_2, and T_1 traverses the arc from S_{11} to S_3, as well as the arc from S_3 to S_{11}. T_1 can delay T_2 both in the forward as well as reverse directions, and we need to account for two units of delay, which is added to the accumulator term.

Step 4: $A_{11}\ LOOP\ A_3 = A_{3'}$

$$A_{3'} = \left(\begin{array}{c|cc} & T_1 & T_2 \\ \hline T_1 & (1,0) & (1,2) \\ T_2 & (0,0) & (1,0) \\ T_3 & (0,0) & (0,0) \\ & \cdots\cdots\cdots\cdots & \\ & 4 & 3 \end{array}\right)$$

This leaves us with two nodes $S_{3'}$ and $S_{4'}$ with two arcs connecting them, one traversed by T_1 and the other by T_2, as shown in Fig. 10(e). We can now split node $S_{3'}$ into two nodes S_{31} and S_{32} one for each of the out-going arcs from $S_{3'}$.

Step 5: $SPLIT(S_{3'}, \{T_1\}) \Rightarrow A_{31}, A_{32}$

$$A_{31} = \left(\begin{array}{c|c} & T_1 \\ \hline T_1 & (1,0) \\ T_2 & (0,0) \\ T_3 & (0,0) \\ & \cdots\cdots \\ & 4 \end{array}\right), \qquad A_{32} = \left(\begin{array}{c|c} & T_2 \\ \hline T_1 & (0,3) \\ T_2 & (1,0) \\ T_3 & (0,0) \\ & \cdots\cdots \\ & 3 \end{array}\right)$$

This leaves us with the task graph shown in Fig. 10(f). We can now independently PIPE S_{31} and S_{32} with $S_{4'}$, to get S_{final}.

Step 6: A_{31} *PIPE* $A_{4'} = A_{4''}$, A_{32} *PIPE* $A_{4''} = A_{final}$

$$A_{final} = \left(\begin{array}{c|ccc} & T_1 & T_2 & T_3 \\ \hline T_1 & (1,0) & (1,3) & (1,2) \\ T_2 & (0,0) & (1,0) & (1,1) \\ T_3 & (0,0) & (0,0) & (1,0) \\ & \cdots\cdots\cdots\cdots\cdots & & \\ & 5 & 4 & 3 \end{array}\right)$$

Adding the max-terms to the accumulator-terms, we get,

$$A_{final} = \left(\begin{array}{c|ccc} & T_1 & T_2 & T_3 \\ \hline T_1 & (0,1) & (0,4) & (0,3) \\ T_2 & (0,0) & (0,1) & (0,2) \\ T_3 & (0,0) & (0,0) & (0,1) \\ & \cdots\cdots\cdots\cdots\cdots & & \\ & 5 & 4 & 3 \end{array}\right)$$

With this final equivalent single stage matrix, we can construct a uniprocessor task set and use any uniprocessor schedulability test to analyze the schedulability of a task in the distributed system. The reduction process and schedulability analysis is similar to the description in Jayachandran and Abdelzaher (2008a), and is omitted in the interest of brevity.

10 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. Further, we introduced an operator called the LOOP to extend the Delay Composition Algebra developed in prior work, to handle non-acyclic distributed systems as well. 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.

Appendix: Proof of correctness of the algebra

We now prove the correctness of the delay composition algebra. That is, if the job is schedulable in the final equivalent uniprocessor task set, then the corresponding job

in the original distributed system is also schedulable. We show that by successively applying the operators of the algebra, the final single stage operand computes the expression for the end-to-end delay bound as per the delay composition theorem. We extend the proof in Jayachandran and Abdelzaher (2008a) for this purpose. The delay composition theorem applied to a job J_k and the set S of higher priority job-segments J_i^s that share a sequence of consecutive common execution stages with J_k is as follows:

$$Delay(J_k) \leq \sum_i \sum_{J_i^s \in S} 2C_{i,max}^s + \sum_{j \in p_k} Node_{j,max} \tag{7}$$

The above inequality can be rewritten as follows:

$$Delay(J_k) \leq \sum_i 2r_{i,k}^* + s_k^* \tag{8}$$

$$r_{i,k}^* = \sum_{J_i^s \in S} C_{i,max}^s;\ s_k^* = \sum_{j \in p_k} Node_{j,max} \tag{9}$$

Let $r_{i,k}^M$ denote the $(i,k)^{\text{th}}$ element in the final single stage matrix derived using the algebra. Since delays due to higher priority jobs are additive on a uniprocessor, the delay that the transformed job J_k, called J_k^*, experiences on the hypothetical uniprocessor is precisely $Delay(J_k^*) = \sum_i 2r_{i,k}^M + s_k^M$ (after multiplying $r_{i,k}^M$ by 2 as per rules in Sect. 8.3). If $r_{i,k}^M = r_{i,k}^*$ and $s_k^M = s_k^*$, it follows that $Delay(J_k) \leq Delay(J_k^*)$. Thus, if J_k^* is schedulable on the uniprocessor, so is J_k in the original distributed system. In the case of periodic tasks, as observed in Jayachandran and Abdelzaher (2008a), finding the actual number of invocations, $Invoc_i$, for each higher priority periodic task, T_i, that delays J_k, is not the responsibility of the algebra or the reduction process. This is handled by the uniprocessor analysis used. The number of invocations, $Invoc_i$, as determined by the uniprocessor analysis will at least be as large as the number of actual invocations of T_i that delay J_k in the distributed system. The reason is because, every invocation of T_i^* that arrives before J_k^* completes execution will delay J_k^* on the uniprocessor, but the corresponding invocations of T_i may never catch-up with J_k to preempt it in the distributed system, as they may be executing on different resources.

We shall now show that in the final matrix, $r_{i,k}^M = r_{i,k}^*$ and $s_k^M = s_k^*$. It is safe to assume that all necessary CUT operations are performed first, as any CUT operation only relaxes precedence constraints and performs a safe transformation of the system that does not improve schedulability. Now, consider the entire sequence of PIPE, SPLIT, and LOOP operations performed to reduce the distributed system to a single node. Let us denote each arc using an unique identifier, and let the set of all arcs in the original distributed system be denoted by L^0. Note that SPLIT operations neither add nor remove arcs from L^0. PIPE operations remove precisely one arc from L^0, and LOOP operations may remove at most two arcs (connecting the same two nodes) from L^0.

In order to compute $r_{i,k}^M$, consider the path of J_k. Let L_k^0 denote the subset of arcs in L^0 that lie on the path of J_k (including arcs in the opposite direction as J_k). As in the proof presented in Jayachandran and Abdelzaher (2008a), all PIPE operations

can be classified under three categories: *path* PIPEs (applied to an arc in L_k^0), *incident* PIPEs (applied to an arc that shares one node with an arc in L_k^0), and *detached* PIPEs (applied to an arc that shares no nodes with arc in L_k^0). SPLIT operations can be classified into two categories: *path* SPLITs (applied to a node with an arc in L_k^0) and *detached* SPLITs (the rest). Likewise, LOOP operations can be classified into *path*, *incident*, and *detached* LOOPs. Trivially, only path PIPEs, path SPLITs, and path LOOPs affect elements in column k of the operand matrices, that is, the components of the delay of job J_k.

Consider a job J_i of higher priority than J_k. Let us denote the set of arcs in $Seg_{i,k}$ (those arcs traveled by both J_i and J_k) as $L_{i,k}^0$. Path PIPEs and path LOOPs that reduce arcs not traveled by J_i simply propagate $q_{i,k}$ of the downstream node, and the sum of the $r_{i,k}$'s of the upstream and downstream nodes to the resultant matrix. This is because, as J_i does not travel the reduced arc it does not execute on the upstream node and $q_{i,k}$ of the upstream node must be zero. The $r_{i,k}$ values in the upstream and downstream nodes denote the delay of independent job-segments of J_i which need to be added together. Further, SPLITs of nodes with no arcs traveled by J_i do not alter $q_{i,k}$ and $r_{i,k}$, since J_i could not have parted J_k at the split node. Hence, we now need to only consider PIPE, LOOP, and SPLIT operations involving arcs traveled by both jobs (that is, in $Seg_{i,k}$).

Consider a segment $s \in Seg_{i,k}$. Let E_s be the last node in the segment. Initially, each node $j \in s$ has $q_{i,k}$ set to the maximum computation time of J_i over all its visits to stage j. To reduce each arc in the segment, a PIPE or LOOP operation must have been performed, producing $q_{i,k}$ to be equal to the maximum of all the computation times of J_i over all the stages in the segment. Recall that a LOOP operation is performed on an arc only when the set of jobs that traverse the arc is a super set of the set of jobs that traverse other outgoing arcs from the node. This ensures that there are no jobs that split from the path of jobs following the arc on which the LOOP operation is performed. Further, note that for each stage, by taking $q_{i,k}$ to be the maximum computation time of J_i over all its visits to the stage, we overestimate the delay of J_k as compared to the delay composition theorem. This is essential, as there is no information stored in the operand matrix with regard to which visit of J_i corresponds to the current segment, and it is safe to consider the maximum computation time over all the visits to the stage. Any SPLIT operation performed on nodes $j \in s$, other than the last node and involving J_i, does not affect $q_{i,k}$ or $r_{i,k}$, as J_k and J_i did not part ways at stage j. Only LOOP and SPLIT operations involving E_s affect the value of $r_{i,k}$.

A LOOP operation that involves E_s and marks the end of the segment s (i.e., removes the last arc that is part of segment s from the task graph), causes $r_{i,k}$ to be updated by adding $q_{i,k}$ to it, which by now equal to the maximum of all the computation times of J_i over all the stages in the segment. If J_i traverses both the forward and reverse arc between the two nodes, then J_i could potentially delay J_k twice, and twice the value of $q_{i,k}$ needs to be added to $r_{i,k}$. If J_i loops back and a new segment begins at one of the two nodes involved, then $q_{i,k}$ is reset to denote the maximum computation time of J_i on that node. At node E_s, any SPLIT operation that splits J_i from J_k can be performed only when there is no incoming arc into E_s that is traversed by J_i. This implies that all PIPE and LOOP operations have been applied over all the other nodes in the segment. Hence, at E_s, $q_{i,k} \geq C_{i,max}^s$ ($q_{i,k}$ may be larger than

$C^s_{i,max}$ as the maximum computation time of J_i over all visits on all stages is taken for each stage operand matrix). The SPLIT would then add $q_{i,k}$ to $r_{i,k}$. As noted before, subsequent operations propagate $r_{i,k}$ to the result node. When all the segments have been reduced, $C^s_{i,max}$ for each segment s is added on to $r_{i,k}$, resulting in $\sum C^s_{i,max}$ over all job-segments J^s_i. In other words, $r^M_{i,k} = r^*_{i,k}$. (Observe that, if J_k and J_i have the same end node, there would be no SPLIT for the last segment and its max-term would still be stored in $q_{i,k}$; this is why we need to compensate for the missing SPLIT and manually add $q^M_{i,k}$ and $r^M_{i,k}$ at the end.)

Similarly, to compute s^M_k, observe that initially for each node j on path p_k, $s^j_k = Node_{j,max}$. Since SPLITs do not affect s_k and PIPEs and LOOPs add it, when all arcs on L^0_k are reduced, $s^M_k = \sum_{j \in p_k} Node_{j,max} = s^*_k$.

References

Audsley AN, Burns A, Richardson M, Tindell K (1993) Applying new scheduling theory to static priority pre-emptive scheduling. Softw Eng 284–292

Cruz R (1991a) A calculus for network delay, part I: Network elements in isolation. IEEE Trans Inf Theory 37(1):114–131

Cruz R (1991b) A calculus for network delay, part II: Network analysis. IEEE Trans Inf Theory 37(1):132–141

Fohler G, Ramamritham K (1997) Static scheduling of pipelined periodic tasks in distributed real-time systems. In: Euromicro workshop on real-time systems, June 1997, pp 128–135

Jayachandran P, Abdelzaher T (2007) A delay composition theorem for real-time pipelines. In: ECRTS, July 2007, pp 29–38

Jayachandran P, Abdelzaher T (2008a) Delay composition algebra: A reduction-based schedulability algebra for distributed real-time systems. In: RTSS, December 2008, pp 259–269

Jayachandran P, Abdelzaher T (2008b) Delay composition in preemptive and non-preemptive real-time pipelines. Real-Time Syst J 40(3):290–320. Special Issue on ECRTS'07

Jayachandran P, Abdelzaher T (2008c) Transforming acyclic distributed systems into equivalent uniprocessors under preemptive and non-preemptive scheduling. In: ECRTS, July 2008, pp 233–242

Jonsson B, Perathoner S, Thiele L, Yi W (2008) Cyclic dependencies in modular performance analysis. In: ACM international conference on embedded software (EMSOFT), October 2008, pp 179–188

Kao B, Garcia-Molina H (1997) Deadline assignment in a distributed soft real-time system. IEEE Trans Parallel Distrib Syst 8(12):1268–1274

Koubaa A, Song Y-Q (2004) Evaluation and improvement of response time bounds for real-time applications under non-preemptive fixed priority scheduling. Int J Prod Res 42(14):2899–2913

Liu CL, Layland JW (1973) Scheduling algorithms for multiprogramming in a hard-real-time environment. J ACM 20(1):46–61

Natale MD, Stankovic JA (1994) Dynamic end-to-end guarantees in distributed real-time systems. In: Proc. real-time systems symposium, December 1994, pp 216–227

Palencia J, Harbour M (2003) Offset-based response time analysis of distributed systems scheduled under EDF. In: Euromicro conference on real-time systems, July 2003, pp 3–12

Pellizzoni R, Lipari G (2005) Improved schedulability analysis of real-time transactions with earliest deadline scheduling. In: RTAS, March 2005, pp 66–75

Thiele L, Chakraborty S, Naedele M (2000) Real-time calculus for scheduling hard real-time systems. In: IEEE international symposium on circuits and systems, vol 4, May 2000, pp 101–104

Tindell K, Clark J (1994) Holistic schedulability analysis for distributed hard real-time systems. Microprocess Microprogram 40(2–3):117–134

Wandeler E, Maxiaguine A, Thiele L (2004) Quantitative characterization of event streams in analysis of hard real-time applications. In: IEEE real-time and embedded technology and applications symposium (RTAS), May 2004, pp 450–459

Xu J, Parnas D (1993) On satisfying timing constraints in hard real-time systems. IEEE Trans Softw Eng 19(1):70–84

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

Praveen Jayachandran received his B.Tech and M.Tech dual degree in Computer Science from the Indian Institute of Technology, Madras, India in 2005. He is currently a PhD student at the University of Illinois at Urbana-Champaign, USA. His research interests include real-time and distributed systems, sensor networks, and wireless networks. He is a recipient of the Andrew and Shana Laursen fellowship awarded by the University of Illinois at Urbana-Champaign, the Vodafone fellowship, the best student paper award at the Euromicro Conference on Real-Time Systems in 2007, and the best paper award at the same conference in 2009. He has received the C.L. and Jane Liu award in 2008, and the Feng Chen Memorial award in 2010, both from the Department of Computer Science at the University of Illinois at Urbana-Champaign.

Tarek Abdelzaher received his B.Sc. and M.Sc. degrees in Electrical and Computer Engineering from Ain Shams University, Cairo, Egypt, in 1990 and 1994 respectively. He received his Ph.D. from the University of Michigan in 1999 on Quality of Service Adaptation in Real-Time Systems. He has been an Assistant Professor at the University of Virginia, where he founded the Software Predictability Group. He is currently an Associate Professor at the Department of Computer Science, the University of Illinois at Urbana Champaign. He has authored/coauthored three book chapters and more than 80 refereed publications in leading conferences and journals in several fields including real-time computing, distributed systems, sensor networks, and control. He is Editor-in-Chief of the Journal of Real-Time Systems, an Associate Editor of the IEEE Transactions on Mobile Computing, IEEE Transactions on Parallel and Distributed Systems, ACM Transaction on Sensor Networks, the International Journal of Embedded Systems and the Ad Hoc Networks Journal, as well as Editor of ACM SIGBED Review. He also held several conference organization positions including Program Chair of RTAS 2004, General Chair of RTAS 2005, Program Chair of RTSS 2006, General Chair of IPSN 2007, General Chair of RTSS 2007, General Chair of Sensys 2008, and General Chair of DCoSS 2008. Abdelzaher's research interests lie broadly in understanding and controlling the temporal properties of software systems in the face of increasing complexity, distribution, and degree of embedding in an external physical environment. Tarek Abdelzaher is a member of IEEE and ACM.