Delay Composition Algebra: A Reduction-based Schedulability Algebra for Distributed Real-Time Systems

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

Abstract

This paper presents the delay composition algebra*: a set of simple operators for systematic transformation of distributed real-time task systems into single-resource task systems such that schedulability properties of the original system are preserved. The transformation allows performing schedulability analysis on distributed systems using uniprocessor theory and analysis tools. Reduction-based analyses techniques have been used in other contexts such as control theory and circuit theory, by defining rules to compose together components of the system and reducing them into equivalent single components that can be easily analyzed. This paper is the first to develop such reduction rules for distributed real-time systems. By successively applying operators such as PIPE and SPLIT on operands that represent workload on composed subsystems, we show how a distributed task system can be reduced to an equivalent single resource task set from which the end-to-end delay and schedulability of tasks can be inferred. We show through simulations that the proposed analysis framework is less pessimistic with increasing system scale compared to traditional approaches.*

1. Introduction

In this paper, we present an algebra for schedulability analysis of distributed real-time systems. The algebra reduces the distributed system workload to an equivalent uniprocessor workload that can be analyzed using uniprocessor schedulability analysis techniques to infer end-to-end delay and schedulability properties of each of the original distributed jobs. Existing techniques for analyzing delay and schedulability of jobs in distributed systems can be broadly classified into two categories: (i) decomposition-based, and (ii) extension-based. The decomposition-based techniques break the system into multiple subsystems, analyze each subsystem independently using current uniprocessor analysis techniques, then combine the results. The extension-based techniques explore ways to extend current uniprocessor analyses to accommodate distributed tasks and resources. In contrast, we propose to use a third category of techniques for analyzing distributed systems that are based on *reduction* (as opposed to decomposition or extension). Rather than breaking up the problem into subproblems, or extending uniprocessor analyses to more complex systems, we systematically *reduce* the distributed system schedulability problem to a single simple problem on a uniprocessor.

A reduction-based approach to system analysis has been devised in many contexts outside distributed system scheduling. For instance, in control theory, there are rules to reduce complex block diagrams into a single equivalent block, which can later be analyzed for stability and performance properties. In circuit theory, laws such as Kirchoff's laws enable complex circuits to be reduced to a single equivalent source and impedance. Apart from reducing the complexity of the problem to that of a single component, such reduction rules also provide fundamental insights into how key performance properties are affected by the structure and arrangement of individual components in the system. The contribution of this paper lies in developing a compositional algebraic framework for analyzing timing issues in distributed systems by reducing them to an equivalent uniprocessor.

While the theory developed in this paper can handle a general class of distributed systems, it has several limitations that motivate further work. For instance, in this paper, we assume that jobs require only a single resource at any given time (although they may need different resources at different times). We do not consider jobs that simultaneously require two or more resources. Further, we only consider systems where a job has the same priority on all resources. Finally, the analysis is developed mostly for acyclic systems (where no cycles in execution graphs exist). We discuss extensions to cyclic systems but do not evaluate them in this work. We hope that this paper serves as one of the first of a series of reduction-based approaches to analyzing complex systems, and that future work can relax some of the restrictive assumptions made.

The rest of this paper is organized as follows. We present the algebra and the intuition behind it in Section 2. In Section 3, we formally prove the correctness of the algebra. We extend our results to non-acyclic systems in Section 4. In Section 5, we evaluate the performance of our algebraic framework through simulation studies. In Section 6, we discuss related work and conclude in Section 7.

2. Delay Composition Algebra

The main goal of the delay composition algebra is to allow schedulability analysis of distributed jobs by reducing them to a uniprocessor workload. Given a graph of system resources, where nodes represent processing resources and arcs represent the direction of job flow, our algebraic opera-

*This work was funded in part by NSF grants CNS 05-53420, CNS 06-13665, and CNS 07-20513

1052-8725/08 $25.00 © 2008 IEEE
DOI 10.1109/RTSS.2008.38

tors systematically "merge" resource nodes, composing their workloads per rules of the algebra, until only one node remains. The workload of that node represents a uniprocessor job set. Uniprocessor schedulability analysis can then be used to determine the schedulability of the set. In this section, we provide a detailed description of the algebra and its underlying basic intuition. A formal proof of correctness is presented in Section 3.

Section 2.1 describes the assumed system model. We provide the intuition leading to our algebra in Section 2.2. In Section 2.3, we describe the basic operand representation and show how to translate a system into operands of the delay composition algebra. The operators of the algebra and a proof of liveness are described in Section 2.4. In Section 2.5, we show how end-to-end delay and schedulability of jobs are determined from the final operand matrix. Finally, we conclude with an illustrative example, in Section 2.6.

2.1 System Model and Problem Statement

Consider a distributed system given by a resource graph of N nodes that serves a set of real-time jobs. Each node in the graph is a resource (e.g., a stage of distributed processing). Arcs represent the direction of execution flow. A resource could be anything that is allocated to jobs in priority order. For example, it could be a processor or a communication link. In this paper, we assume that a job has the same priority on all resources of the distributed system. We focus first on acyclic systems, meaning that the resource graph is a directed acyclic graph (DAG). Later, in Section 4, we show that if cycles or bi-directional arcs exist between nodes, we can cut the offending arcs using a task set transformation, reducing the graph to a DAG again. Different jobs in the system can traverse different paths in the DAG, and may have different start and end nodes. Each job traverses a sequence of multiple stages of execution, between its start and end node, and must exit the system within a pre-specified end-to-end deadline. The DAG is the union of all job paths. Let the (worst-case) execution time of job J_i on stage j in its path be denoted by $C_{i,j}$, and let the relative end-to-end job deadline be denoted by D_i.

We further augment the DAG with an arc from each end node of a job to a single virtual "finish" node, f. The execution time of any job J_i on the finish node f, $C_{i,f}$, is set to zero, so as to not affect schedulability. This augmentation ensures that the graph is never partitioned and hence can be reduced to a single node using our algebraic operators.

We make no assumptions on the periodicity of the task set. Indeed, jobs may or may not be invocations of periodic tasks. In our model, if the jobs are in fact invocations of periodic tasks, different invocations of the same periodic task can simultaneously be present in the system, and can have different priority values. The above model can represent static and dynamic priority scheduling, periodic or aperiodic task scheduling, as well as preemptive and non-preemptive scheduling. The question we would like to answer is whether each job is schedulable (i.e., can traverse its path through the system by its deadline).

2.2 Intuition for a Reduction Approach

To answer the above question, we reduce the distributed system to a single node. Our reduction operators simplify the resource DAG progressively by breaking forks into chains and compacting chains by merging neighboring nodes, producing an equivalent workload for the resulting merged node. Workload of any one node (that may represent a single resource or the result of reducing an entire subsystem) is described generically by a two-dimensional matrix stating the worst-case delay that each job, J_i, imposes on each other job, J_k, in the subsystem the node represents. Let us call it the *load matrix* of the subsystem in question.

Observe that if jobs are invocation instances of periodic or sporadic tasks (which we expect to be the most common use of our algebra), we include in the load matrix only one instance of each task. We need to consider only one instance of each task because all individual invocation instances of the same task have the same parameters and thus will impose the same delay on a lower priority instance. It is therefore enough to compute this delay once. We are able to get away with this because our algebra is only concerned with job transformation. It is not concerned, for example, with computing the number of invocations of one task that may preempt another. This is the responsibility of uniprocessor schedulability analysis that we apply to the resulting uniprocessor task set. The algebra simply reduces a distributed instance into a uniprocessor instance. This decoupling between the reduction part and the analysis part is a key advantage of the reduction-based approach. Hence, in the following, when we mention a job, it could either mean an aperiodic job or a single representative instance of a periodic or sporadic task. For periodic or sporadic task sets, the dimension of the load matrix is therefore $n \times n$, where n is the finite number of *tasks* in the set.

Observe that, on a node that represents a single resource j, any job J_i, that is of higher priority than job J_k, can delay the latter by at most J_i's worst-case computation time, $C_{i,j}$, on that resource. This allows one to trivially produce the load matrix for a single resource given job computation times, $C_{i,j}$, on that resource. Element (i,k) of the load matrix for resource j, denoted $q^j_{i,k}$ (or just $q_{i,k}$ where no ambiguity arises) is equal to $C_{i,j}$ as long as J_i is of (equal or) higher priority than J_k. It is zero otherwise.

The main question becomes, in a distributed system, how to compute the worst-case delay that a job imposes on another when the two meet on more than one resource? The answer decides how delay components of two load matrices are combined when the resource nodes corresponding to these matrices are merged using our algebraic operators. Intuitions derived from uniprocessor systems suggest that delays are combined *additively*. This is not true in distributed systems. In particular, we have shown in [7] that delays in pipelines are *sub-additive* because of gains due to parallelism caused by pipelining. More specifically, the worst-case delay imposed by a higher priority job, J_i, on a lower priority job, J_k, when both traverse the same set of stages, varies with the

maximum of J_i's per-stage computation times, not their *sum* (plus another component we shall mention shortly).

The delay composition algebra leverages the aforementioned result. Neighboring nodes in the resource DAG present an instance of pipelining, in that jobs that complete execution at one node move on to execute at the next. Hence, when these neighboring nodes are combined, the delay components, $q_{i,k}$, in their load matrices are composed by a maximization operation. In our algebra, this is done by the PIPE operator. It reduces two neighboring nodes to one and combines the corresponding elements, $q_{i,k}$, of their respective load matrices by taking the maximum of each pair. For this reason, we call $q_{i,k}$ the *max term*.

It could be, however, that two jobs travel together in a pipelined fashion[1] for a few stages (which we call a pipeline segment), then split and later merge again for several more stages (i.e., another pipeline segment). Figure 1 demonstrates such a scenario for a job J_k and a higher priority job, J_i. In this case, the max terms of each of the pipeline segments (computed by the maximization operator) must be added up to compute the total delay that J_i imposes on J_k. It is convenient to use a running counter or "accumulator" for such addition. Whenever the jobs are pipelined together, delays are composed by maximization (kept in the max term) as discussed above. Every time J_i splits away from J_k, signaling the termination of one pipeline segment, the max term (i.e., the delay imposed by J_i on J_k in that segment) is added to the accumulator. Let the accumulator be denoted by $r_{i,k}$. Hence, $r_{i,k}$ represents the total delay imposed by J_i on a lower priority job J_k over all past pipeline segments they shared. Observe that jobs can split apart only at those nodes in the DAG that have more than one outgoing arc. Hence, in our algebra, a SPLIT operator is used when a node in the DAG has more than one outgoing arc. SPLIT updates the respective accumulator variables, $r_{i,k}$, of all those jobs J_k, where J_k and a higher priority job J_i part on different arcs. The update simply adds $q_{i,k}$ to $r_{i,k}$ and resets $q_{i,k}$ to zero.

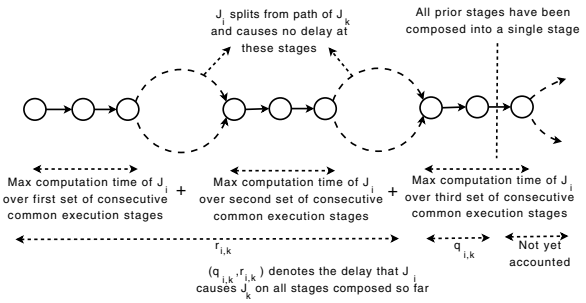

Figure 1. Figure showing the components of the delay that J_i causes J_k, and how the composition of stages works

In summary, in a distributed system, it is useful to represent the delay that one job J_i imposes on another J_k as the sum of two components $q_{i,k}$ and $r_{i,k}$. The $q_{i,k}$ term is updated upon PIPEs using the maximization operator (the max term). The $r_{i,k}$ is the accumulator term. The $q_{i,k}$ is added to the $r_{i,k}$ (and reset) upon SPLITs, when J_i splits from the path of J_k. PIPE and SPLIT are thus the main operators of our algebra. In the final resulting matrix, the $q_{i,k}$ and $r_{i,k}$ components are added to yield the total delay that each job imposes on another in the entire system.

The final matrix is indistinguishable from one that represents a uniprocessor task set. In particular, each *column* k in the final matrix denotes a uniprocessor set of jobs that delay J_k. In this column, each non-zero element determines the computation time of one such job J_i. Since the transformation is agnostic to periodicity, for periodic tasks, J_i and J_k simply represent the parameters of the corresponding periodic task invocations. Hence, for any task, T_k, in the original distributed system, the final matrix yields a uniprocessor task set (in column k), from which the schedulability of task T_k can be analyzed using uniprocessor schedulability analysis.

Finally, the above discussion omitted the fact that the results in [7] also specified a component of pipeline delay that grows with the number of stages traversed by a job and is independent of the number of higher priority jobs. We call it the *stage-additive* component, s_k. Hence, the load matrix, in fact, has an extra row to represent this component. As the name suggests, when two nodes are merged, this component is combined by addition. With the above background and intuition in mind, in the following subsections, we describe the algebra more formally, then prove it.

2.3 Operand Representation

In order to represent a task set on a resource for the purpose of analyzing delay and schedulability, we represent the delay that each job (or periodic task invocation) causes every other job in the system. As mentioned above, we represent this as an $n \times n$ array of delay terms, with the $(i,k)^{th}$ element denoting the delay that job J_i causes job J_k. Each element (i,k) is represented as a two-tuple $(q_{i,k}, r_{i,k})$, where the first term in the tuple $q_{i,k}$ denotes the max-term, and the second term $r_{i,k}$ denotes the accumulator-term. The operand matrix has an additional row in which the k^{th} element, s_k (that we shall define shortly), represents the delay of job J_k that is independent of the number of jobs in the system, and is additive across the stages on which J_k executes. An operand A, represented as an $(n+1) \times n$ matrix is shown below:

$$A = \left(\begin{array}{c|cccc} & J_1 & J_2 & \ldots & J_n \\ \hline J_1 & (q^A_{1,1}, r^A_{1,1}) & (q^A_{1,2}, r^A_{1,2}) & \ldots & (q^A_{1,n}, r^A_{1,n}) \\ J_2 & (q^A_{2,1}, r^A_{2,1}) & (q^A_{2,2}, r^A_{2,2}) & \ldots & (q^A_{2,n}, r^A_{2,n}) \\ . & . & . & & \\ . & . & & & \\ . & . & & & \\ J_n & (q^A_{n,1}, r^A_{n,1}) & (q^A_{n,2}, r^A_{n,2}) & \ldots & (q^A_{n,n}, r^A_{n,n}) \\ & \cdots & \cdots & \cdots & \cdots \\ & s^A_1 & s^A_2 & \ldots & s^A_n \end{array}\right)$$

[1] The term *pipelined execution* has also been used in the literature to refer to the situation where an invocation of a task can start before the previous invocation has completed, when deadlines are larger than task periods. We do not intend the term pipelined execution in this context.

Let us now construct the matrix for a single stage j, the basic operand. Let us first assume that the scheduling is preemptive. Without loss of generality, assume that the jobs are ordered according to priority, and $i < k$ implies that J_i has a higher priority than J_k. If a job J_k does not execute at stage j, then all the elements $q_{i,k}$, $r_{i,k}$, and s_k are set to zero. For each job J_k that executes on stage j, $r_{i,k} = 0$ (the accumulator is initialized to zero). Further, $q_{i,k} = C_{i,j}$ if J_i executes on stage j and has a higher or equal priority compared to J_k. Otherwise, $q_{i,k} = 0$. The term s_k, is defined as the maximum computation time over all higher priority jobs on stage j. That is, $s_k = \max_{i \le k} C_{i,j}$. An example operand matrix for a stage j in a system with four jobs, of which only J_1, J_2 and J_4 execute on the stage, is shown below:

	J_1	J_2	J_3	J_4
J_1	$(C_{1,j}, 0)$	$(C_{1,j}, 0)$	$(0,0)$	$(C_{1,j}, 0)$
J_2	$(0,0)$	$(C_{2,j}, 0)$	$(0,0)$	$(C_{2,j}, 0)$
J_3	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$
J_4	$(0,0)$	$(0,0)$	$(0,0)$	$(C_{4,j}, 0)$
	.....			
	$C_{1,j}$	$max(C_{1,j}, C_{2,j})$	0	$max(C_{1,j}, C_{2,j}, C_{4,j})$

When the scheduling is non-preemptive, the matrix for a single stage j is constructed similarly, except for the stage-additive component s_k, which in this case is a sum of two terms. The first term is the maximum computation time of *any* job (not just higher priority jobs) on stage j. The second term is the maximum computation time of any lower priority job on stage j. Thus, $s_k = \max_i C_{i,j} + \max_{i>k} C_{i,j}$. An example matrix for a stage j under non-preemptive scheduling, for the same 4-job system as before is shown below:

	J_1	J_2	J_3	J_4
J_1	$(C_{1,j}, 0)$	$(C_{1,j}, 0)$	$(0,0)$	$(C_{1,j}, 0)$
J_2	$(0,0)$	$(C_{2,j}, 0)$	$(0,0)$	$(C_{2,j}, 0)$
J_3	$(0,0)$	$(0,0)$	$(0,0)$	$(0,0)$
J_4	$(0,0)$	$(0,0)$	$(0,0)$	$(C_{4,j}, 0)$
	.....			
	$C_{1,j} + max(C_{2,j}, C_{4,j})$	$max(C_{1,j}, C_{2,j}) + C_{4,j}$	0	$max(C_{1,j}, C_{2,j}, C_{4,j})$

2.4 Operators of the Algebra

We describe the two operators, namely PIPE and SPLIT. These operators ensure that every term $(q_{i,k}, r_{i,k})$ in the resultant operand matrix correctly represents the max-term and accumulator-term of the delay that J_i can cause J_k over all the stages that the operand represents.

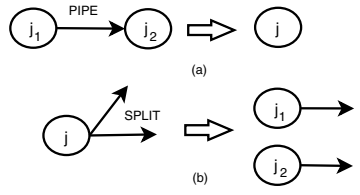

Figure 2. Figure showing the operators and the equivalent stages they result in (a) PIPE (b) SPLIT

2.4.1 The PIPE Operator

The PIPE operator merges two neighboring nodes in the resource graph (as shown in Figure 2(a)). Each of the two nodes being merged may themselves be resulting from the composition of multiple nodes. PIPE can be applied to any two nodes connected by an arc as long as the node at the tail of the arc (i.e., the upstream node) has only one outgoing arc. If the node has more than one outgoing arc, it must be split first as described in the SPLIT operator.

Let $C = A\ PIPE\ B$, where A, B, and C are matrices of the form described in Section 2.3. The result of the PIPE operation $(q^C_{i,k}, r^C_{i,k})$ is obtained by taking the maximum of corresponding elements $q_{i,k}$ and $r_{i,k}$ from the two operand matrices A and B. As we shall show later in Section 3, only the first (i.e., upstream) of the elements $r_{i,k}$ from the two operand matrices can be non-zero, so the max operation on the $r_{i,k}$ elements essentially copies the upstream value of $r_{i,k}$ onto matrix C. The stage-additive component, s^C_k, on the other hand is additive across stages, and hence the corresponding stage-additive components from the two operand matrices are added. The PIPE operator can formally be defined as follows:

Definition 1: 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, $C = A\ PIPE\ B$, as follows:

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

□

For instance, when jobs J_1 and J_2 execute on stages 1 and 2 (represented as matrices A and B, respectively), the PIPE operation between the two stages can be denoted as:

	J_1	J_2
J_1	$(q^A_{1,1}, r^A_{1,1})$	$(q^A_{1,2}, r^A_{1,2})$
J_2	$(0,0)$	$(q^A_{2,2}, r^A_{2,2})$
	.....	
	s^A_1	s^A_2

$PIPE$

	J_1	J_2
J_1	$(q^B_{1,1}, r^B_{1,1})$	$(q^B_{1,2}, r^B_{1,2})$
J_2	$(0,0)$	$(q^B_{2,2}, r^B_{2,2})$
	.....	
	s^B_1	s^B_2

$=$

	J_1	J_2
J_1	$(max(q^A_{1,1}, q^B_{1,1}),$ $max(r^A_{1,1}, r^B_{1,1}))$	$(max(q^A_{1,2}, q^B_{1,2}),$ $max(r^A_{1,2}, r^B_{1,2}))$
J_2	(0,0)	$(max(q^A_{2,2}, q^B_{2,2}),$ $max(r^A_{2,2}, r^B_{2,2}))$
	.....	
	$s^A_1 + s^B_1$	$s^A_2 + s^B_2$

2.4.2 The SPLIT Operator

The SPLIT operator can be used when a node in the resource graph has more than one outgoing arc, but no incoming arcs. (If the node has incoming arcs, PIPE should be executed first along such arcs.) The existence of multiple outgoing arcs indicates that the paths of some jobs split from each other

and move on to execute at different stages. Let there be l outgoing arcs at node j. The operator splits that node into l nodes each with only one outgoing arc. The load matrix A of node j is split into l matrices, one for each resulting node/arc. Each of these l matrices is obtained by replicating matrix A and zeroing out the columns corresponding to jobs that do not follow the arc in question. Further, for any job J_k and a higher priority job J_i on node j, if the two jobs continue on different arcs, the accumulator term of J_k must be updated. Hence, in the output matrix containing J_k (where the column elements corresponding to a job J_i are zero), we update elements $(q_{i,k}, r_{i,k})$ with $(0, q_{i,k} + r_{i,k})$. Figure 2(b) shows the SPLIT operation with $l = 2$, and two hypothetical stages are created after the operation. We formally define the SPLIT operator as follows:

Definition 2: SPLIT Operator. Let matrix A represent node j, with $l > 1$ outgoing arcs and no incoming arcs. Let sets X_1, X_2, ..., X_l represent the sets of jobs that flow from j along each of these arcs respectively. Hence, $X_{i_1} \cap X_{i_2} = \phi$ for $i_1 \neq i_2$. The SPLIT operation results in l nodes, each with one outgoing arc and load matrix, $A_x, 1 \leq x \leq l$, obtained as follows:

$\forall J_k$:

1. if $J_k \in X_x$:
   $s_k^{A_x} = s_k^A$; $\forall i$: if $J_i \in X_x$: $q_{i,k}^{A_x} = q_{i,k}^A$, $r_{i,k}^{A_x} = r_{i,k}^A$, else
   $q_{i,k}^{A_x} = 0$, $r_{i,k}^{A_x} = q_{i,k}^A + r_{i,k}^A$.

2. if $J_k \notin X_x$:
   $s_k^{A_x} = 0$; $\forall i$: $q_{i,k}^{A_x} = 0$, $r_{i,k}^{A_x} = 0$.

□

For instance, when jobs J_1 and J_2 executing on stage j (represented by matrix A) split from one another, the resultant matrices are created as shown below:

$$SPLIT \left(\begin{array}{c|cc} & J_1 & J_2 \\ \hline J_1 & (q_{1,1}^A, r_{1,1}^A) & (q_{1,2}^A, r_{1,2}^A) \\ J_2 & (0,0) & (q_{2,2}^A, r_{2,2}^A) \\ & \cdots\cdots\cdots\cdots\cdots\cdots & \\ & s_1^A & s_2^A \end{array} \right) \equiv$$

$$\left(\begin{array}{c|cc} & J_1 & J_2 \\ \hline J_1 & (q_{1,1}^A, r_{1,1}^A) & (0,0) \\ J_2 & (0,0) & (0,0) \\ & \cdots\cdots\cdots\cdots\cdots & \\ & s_1^A & 0 \end{array} \right), \left(\begin{array}{c|cc} & J_1 & J_2 \\ \hline J_1 & (0,0) & (0, q_{1,2}^A + r_{1,2}^A) \\ J_2 & (0,0) & (q_{2,2}^A, r_{2,2}^A) \\ & \cdots\cdots\cdots\cdots\cdots\cdots & \\ & 0 & s_2^A \end{array} \right)$$

Observe that, for a system with n jobs, every operand matrix has $n + 1$ rows and n columns. Any job that does not execute at a stage has a column with all its elements set to zero. It is possible to optimize this representation by removing all zero-element columns and having operand matrices of variable dimensions. Row and column indices would have to be represented explicitly (rather than the implicit global job-numbering assumed in the above exposition).

The definition of the operators are the same regardless of whether the scheduling in the system is preemptive or non-preemptive. By applying these operators in succession, the distributed system can be reduced to an equivalent single stage represented by a single operand matrix. Note that, as both the max and sum operations are commutative and associative, the PIPE operation is commutative and associative as well. Due to space limitations, we do not formally prove these properties, nor show that the final single stage obtained after the reduction procedure is unique regardless of the order in which the operations are performed.

2.4.3 Proof of Liveness and Algorithm Complexity

Given the operator definitions described above, we now prove the following theorem:

Theorem 1: *The delay composition algebra always reduces the original resource DAG (augmented with the extra finish node as mentioned in Section 2.1) to a single node.*

Proof: To prove the above, observe that we defined the following rules for applying the algebraic operators: (i) a PIPE can only be applied to a pair of nodes if the upstream node has exactly one outgoing arc, and (ii) a SPLIT can only be applied to a node if it has no incoming arcs and multiple outgoing arcs.

Hence, a PIPE can always be performed unless we are left only with those nodes that have *multiple* outgoing arcs (and their immediate downstream neighbors). However, in such a case, a SPLIT can always be performed on the earliest of these nodes. This is because (i) this node does not have incoming arcs from earlier nodes (that would contradict it being earliest), and (ii) it has multiple outgoing arcs (since only such nodes are left together with their downstream neighbors but the earliest node, by definition, is not downstream from another). Hence, at any given time, either a PIPE or a SPLIT can always be performed until no arcs are left.

It is left to show that the graph always remains connected, and hence when no arcs are left only one node remains. We prove it by induction. First note that the initial DAG is connected, and the virtual finish node f is downstream from every node. This is because each node j is either an end node of some job, in which case it is connected directly downstream to the virtual finish node, f, or is not an end node, in which case it must have a downstream path to the end node of some job, and the latter is connected downstream to the virtual finish node. Hence, the finish node can be reached from any node by a downstream path and the graph is connected. Next, we prove the induction step, showing that applying a PIPE or SPLIT does not disconnect the graph and keeps f downstream from every node. For a PIPE, this is self-evident, since it only merges nodes. For a SPLIT, assume that the graph before the SPLIT was applied was connected and each node had a downstream path to the virtual finish node. SPLIT takes a node j with an immediate downstream neighbor set N_j and replaces it with multiple nodes, each inheriting a downstream arc to one of these neighbors. Thus, since neighbors in set N_j are connected to the virtual finish node by a downstream path, so will be each of the newly created nodes. The induction hypothesis is maintained. By induction, the graph is never disconnected by PIPEs or SPLITs, and the finish node is always downstream from every node. Hence, when the algebra has removed all arcs, the DAG is reduced to a single node. □

A PIPE operation can be performed in $O(n^2)$ time, where n is the number of jobs in the system, and each PIPE operation reduces the number of arcs in the DAG by one. The time complexity for a SPLIT operation involving k arcs is $O(kn^2)$, and each of these k arcs can be eliminated through PIPE operations in the next step. Hence, the complexity for eliminating each arc is $O(n^2)$, and the net complexity of the algebra to reduce a DAG to a single node is $O(|E|n^2)$, where $|E|$ is the number of arcs in the original resource DAG.

2.5 Task Set Transformation

Once the system is reduced to one node, the end-to-end delay and schedulability of any job J_k can be inferred from the node's load matrix. Remember that in periodic or sporadic task systems, J_k stands for an instance of task T_k. We shall use the task notation in this section, since we expect the algebra to be applied mostly for periodic or sporadic task sets. Once the system is reduced to one node, the max-term for each element in the final matrix is first added to the accumulator term, that is, $(q_{i,k}, r_{i,k})$ is replaced by $(0, q_{i,k}+r_{i,k})$. To analyze the schedulability of any task T_k in the original distributed system, an equivalent uniprocessor task set is obtained from column k of the final load matrix 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 Section 3). The period P_i (if T_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 Section 3 that if T_k^* meets its deadline on the uniprocessor when scheduled together with this 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. First, however, we present an example.

2.6 An Illustrative Example

We now illustrate the composition of a distributed system into an equivalent single stage using the algebra. We consider an eight stage system as shown in Figure 3(a). There are three periodic tasks executing in the system T_1, T_2, and T_3, in decreasing priority order. T_1 follows the path $S_1 - S_3 - S_4 - S_5 - S_7 - S_8$, T_2 follows $S_1 - S_3 - S_6 - S_7 - S_8$, and T_3 follows $S_2 - S_3 - S_6 - S_7 - S_8$. For simplicity, let us assume that every job requires one unit of computation time at each stage in its path. T_1 has a period (same as end-to-end deadline) of 10 time units, while T_2 and T_3 have a period (same as end-to-end deadline) of 20 time units. In describing the matrices, we remove all columns with zero elements for the sake of conciseness. As the example has only one finish-node (S_8), we do not create an additional virtual finish-node.

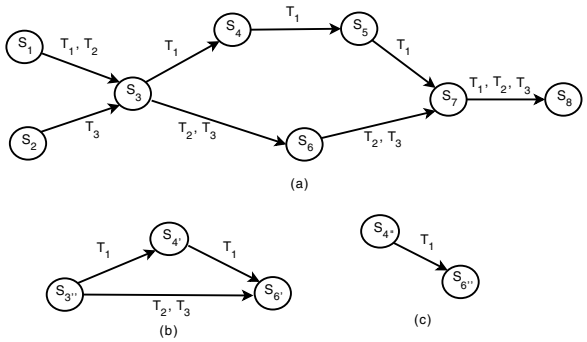

Figure 3. (a) Example system to be composed (b) Composed system after 2 steps (c) Composed system after 4 steps

Let the initial matrices for stage S_i be denoted by A_i. We start the composition with the PIPE operations between S_1 and S_3 to give $S_{3'}$, and PIPE S_2 with $S_{3'}$ resulting in $S_{3''}$.
Step 1: $A_1\ PIPE\ A_3 = A_{3'}$, $A_2\ PIPE\ A_{3'} = A_{3''}$

As the maximum of job-additive components and the sum of stage-additive components are taken, we get,

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

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

We next PIPE S_7 and S_8 to form $S_{7'}$, S_4 and S_5 to form $S_{4'}$, S_6 and $S_{7'}$ to form $S_{6'}$.
Step 2: $A_7\ PIPE\ A_8 = A_{7'}$, $A_4\ PIPE\ A_5 = A_{4'}$, $A_6\ PIPE\ A_{7'} = A_{6'}$

$$A_{7'} = \left(\begin{array}{c|ccc} & T_1 & T_2 & T_3 \\ \hline T_1 & (1,0) & (1,0) & (1,0) \\ T_2 & (0,0) & (1,0) & (1,0) \\ T_3 & (0,0) & (0,0) & (1,0) \\ & \cdots & \cdots & \cdots \\ & 2 & 2 & 2 \end{array}\right)$$

$$A_{4'} = \left(\begin{array}{c|c} & T_1 \\ \hline T_1 & (1,0) \\ & \cdots \\ & 2 \end{array}\right), A_{6'} = \left(\begin{array}{c|ccc} & T_1 & T_2 & T_3 \\ \hline T_1 & (1,0) & (1,0) & (1,0) \\ T_2 & (0,0) & (1,0) & (1,0) \\ T_3 & (0,0) & (0,0) & (1,0) \\ & \cdots & \cdots & \cdots \\ & 2 & 3 & 3 \end{array}\right)$$

The distributed system obtained after steps 1 and 2 is shown in Figure 3(b). Stage $S_{3''}$ does not have any incoming arcs and can be split into stages S_{3_1} consisting of task T_1 (represented as A_{3_1}), and S_{3_2} consisting of tasks T_2 and T_3 (represented as A_{3_2}).
Step 3: $SPLIT(A_{3''}) => A_{3_1}, A_{3_2}$

$$A_{3_1}=\left(\begin{array}{c|c} & T_1 \\ \hline T_1 & (1,0) \\ & \cdots\cdots \\ & 2 \end{array}\right), A_{3_2}=\left(\begin{array}{c|cc} & T_2 & T_3 \\ \hline T_1 & (0,1) & (0.1) \\ T_2 & (1,0) & (1,0) \\ T_3 & (0,0) & (1,0) \\ & \cdots\cdots\cdots\cdots & \\ & 2 & 2 \end{array}\right)$$

Note that as T_1 has split from the paths of the other two jobs, the max-terms of the delay that T_1 causes T_2 and T_3 on all previous stages has been accumulated on to the respective accumulator-terms. We now have only PIPE operations left. We PIPE S_{3_1} with $S_{4'}$ (giving rise to stage $S_{4''}$), and S_{3_2} with $S_{6'}$ (giving rise to stage $S_{6''}$).

Step 4: $A_{3_1}\ PIPE\ A_{4'} = A_{4''}$, $A_{3_2}\ PIPE\ A_{6'} = A_{6''}$

$$A_{4''}=\left(\begin{array}{c|c} & T_1 \\ \hline T_1 & (1,0) \\ & \cdots\cdots \\ & 4 \end{array}\right), A_{6''}=\left(\begin{array}{c|ccc} & T_1 & T_2 & T_3 \\ \hline T_1 & (1,0) & (1,1) & (1,1) \\ T_2 & (0,0) & (1,0) & (1,0) \\ T_3 & (0,0) & (0,0) & (1,0) \\ & \cdots\cdots\cdots\cdots\cdots & & \\ & 2 & 5 & 5 \end{array}\right)$$

Figure 3(c) shows the distributed system after step 4. We finally PIPE $S_{4''}$ and $S_{6''}$ to obtain the single equivalent final stage S_{final} (represented as A_{final}).

Step 5: $A_{4''}\ PIPE\ A_{6''} = A_{final}$, copying max-terms into accumulator-terms

$$A_{final}=\left(\begin{array}{c|ccc} & T_1 & T_2 & T_3 \\ \hline T_1 & (0,1) & (0,2) & (0,2) \\ T_2 & (0,0) & (0,1) & (0,1) \\ T_3 & (0,0) & (0,0) & (0,1) \\ & \cdots\cdots\cdots\cdots\cdots & & \\ & 6 & 5 & 5 \end{array}\right)$$

Using the final matrix A_{final}, we can test the schedulability of each of the three tasks using any single stage schedulability analysis technique, as described in Section 2.5. In this example, we shall use the response time analysis technique proposed in [1]. In the interest of brevity, we shall only analyze the schedulability of T_3.

To test the schedulability of T_3, we construct 3 tasks T_1^* with a computation time of 4 units (twice the value in the column for T_3 and the first row in matrix A_{final}) and period 10, T_2^* with a computation time of 2 (twice the value in the column for T_3 and the second row in matrix A_{final}) and period 20, and T_3^* with a computation time $2 + 4$ (the value in the third row in the column for T_3 + the stage-additive component of T_3) and period 20. In the first iteration of the response time analysis [1], the delay of T_3^* due to one invocation each of T_1^* and T_2^* is calculated as $4 + 2 + 6 = 12$ time units. As this is more than the period of T_1^*, one additional invocation of T_1^* can delay T_3^*, increasing T_3^*'s delay to 16 time units. As this is lower than T_3^*'s deadline of 20 time units, T_3^* is schedulable on the hypothetical uniprocessor. Therefore, T_3 is schedulable in the original distributed system.

3. Proof of Correctness

In this section, we prove the correctness of the delay composition algebra. By correctness, we mean that if a job is schedulable in the resulting uniprocessor task set, it is schedulable in the original distributed system. Below, we show the proof for preemptive systems. The proof for non-preemptive systems is similar and is thus omitted. Consider a job J_k that executes along a path p_k in the original directed acyclic graph. It is desired to determine the schedulability of J_k. Consider a higher-priority job J_i ($i \neq k$) that executes along path p_i. Let paths p_k and p_i intersect in some set $Seg_{i,k}$ of sequences of consecutive (i.e., directly connected) nodes. For example if J_k has the path (1, 2, 5, 8, 11, 13) and J_i has the path (9, 1, 2, 16, 8, 11, 10) then $Seg_{i,k} = \{(1,2),(8,11)\}$. Each member of this set is a shared path segment between J_k and J_i. Let the part of J_i that executes on segment s in set $Seg_{i,k}$ be called sub-job J_i^s. In the above example, J_i^1 is the part of J_i that executes on the path segment (1, 2) and J_i^2 is the part of J_i that executes on the path segment (8, 11). Note that sub-jobs J_i^s are the only parts of J_i that may delay J_k since they are the only parts that share (part of) J_k's path. Let the maximum execution time of sub-job J_i^s over its path be $C_{i,max}^s$. Let the maximum execution time of all jobs J_i^s on node j be $Node_{j,max}$. In an earlier result [7], we derived a delay bound for jobs in a pipelined preemptive system, called the *preemptive pipeline delay composition theorem*. Applied to job J_k and the set S of all sub-jobs J_i^s of all higher-priority jobs that share its path, the theorem states that J_k may be delayed due to these jobs by a total amount, $Delay(J_k)$, bounded as follows:

$$Delay(J_k) \leq \sum_i (\sum_{J_i^s \in S} 2C_{i,max}^s) + \sum_{j \in p_k} Node_{j,max} \quad (1)$$

A proof of this theorem for the case when jobs share all nodes of the path can be found in [7] and extended in [9] to jobs that share arbitrary subsets of the path as in the case above (in view of simplicity, we have chosen a slightly more pessimistic bound than what was derived in [7]). We can rewrite the above inequality as:

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

$$r_{i,k}^* = \sum_{J_i^s \in S} C_{i,max}^s; \quad s_k^* = \sum_{j \in p_k} Node_{j,max} \quad (3)$$

For periodic or sporadic systems, the summation in Inequality (2) can be rewritten in terms of the higher priority tasks (set S_T) and their instances as follows:

$$Delay(T_k) \leq \sum_{T_i \in S_T} 2\ (Invoc_i) r_{i,k}^* + s_k^* \quad (4)$$

where $Invoc_i$ is the number of invocations of task T_i that delay J_k. For example, in deadline-monotonic scheduling, if deadlines are less than periods, $Invoc_i$ is bounded by $\lceil D_k/P_i \rceil$. Since, on a uniprocessor, delays due to higher-priority jobs are additive, adding the delays in column k (after multiplying $r_{i,k}^M$ by 2 per rules in Section 2.5), yields that the transformed J_k, called J_k^*, is delayed on the uniprocessor precisely by $Delay(J_k^*) = \sum_i 2r_{i,k}^M + s_k^M$. Similarly, for periodic or sporadic systems, an invocation of the transformed task T_k, denoted T_k^*, is delayed due to higher-priority

tasks by $Delay(T_k^*) = \sum_i 2(Invoc_i) r_{i,k}^M + s_k^M$. Comparing these expressions to Inequality (2) and (4) it follows that if $r_{i,k}^M = r_{i,k}^*$ and $s_k^M = s_k^*$, then substituting in the delay composition theorem, we get:

$$Delay(J_k) \leq Delay(J_k^*) \quad (5)$$

$$Delay(T_k) \leq Delay(T_k^*) \quad (6)$$

Thus, if J_k^* (T_k^*) is schedulable on the uniprocessor, so is J_k (T_k) in the original system. Observe that finding the actual number, $Invoc_i$, for each T_i is not the responsibility of our reduction. It is the responsibility of schedulability analysis on the reduced set. $Invoc_i$ as determined by uniprocessor schedulability analysis will at least be as large as the number of invocations of T_i that delay T_k in the distributed system. This is because every invocation of T_i^* that arrives before J_k^* (T_k^*) completes execution will delay J_k^* on the uniprocessor, but the corresponding invocation of T_i in the distributed system may never catch up with J_k to preempt it as they may be executing on different resources [8].

We now prove that, indeed, in matrix M, produced by the delay composition algebra for the system at hand, elements of column k satisfy $r_{i,k}^M = r_{i,k}^*$ and $s_k^M = s_k^*$.

To compute the elements of column k of the final result matrix M, consider the entire sequence of PIPE and SPLIT operations used to reduce the original directed acyclic resource graph (augmented with a finish node) to a single node. Since these operations will split or merge nodes in that graph, it is useful to refer to its arcs by unique identifiers, as opposed to their source-destination vertex pairs. Let the set of the unique identifiers given to the arcs in the initial graph be denote by L^0. In the following, for short-hand, we refer to an arc whose identifier is $l \in L^0$ as simply arc l. First, note that SPLITs neither add nor remove arcs in L^0. They simply split a node that has no incoming arcs into multiple nodes, each inheriting one of the original outgoing arcs of the split node. PIPEs remove exactly one arc from L^0.

To compute the $r_{i,k}^M$ entries in column k of the final matrix M, consider the path of job J_k through the graph. Let the subset of arcs in L^0 that lie on this path be denoted by L_k^0. Observe that, all PIPEs performed fall into one of three categories: *path* PIPEs (those applied to an arc in L_k^0), *incident* PIPEs (those applied to an arc that shares only one node with an arc in L_k^0), and *detached* PIPEs (those that share no nodes with arcs in L_k^0). Similarly, for SPLIT, there are two categories: *path* SPLITs (those applied to a node with an arc in L_k^0) and *detached* SPLITs (the rest).

Trivially, detached PIPEs and SPLITs do not change column k of any node because the nodes they are applied to have zeros in column k (not being on J_k's path). It is also easy to see that incident PIPEs do not change column k of the node on the path of J_k, since they max it with zero (because J_k does not execute on the other node). Hence, column k is affected only by path PIPEs and path SPLITs.

Consider one job J_i of higher priority than J_k. Let $L_{i,k}^0 \in L_k^0$ denote the set of arcs in $Seg_{i,k}$ (i.e., those traveled by both J_i and J_k). Note that, path PIPEs reducing arcs not traveled by J_i (i.e., not in $L_{i,k}^0$) simply propagate $q_{i,k}$ of the downstream node and $r_{i,k}$ of the upstream node to the result. This is because $q_{i,k}$ of the upstream node must be zero as J_i does not travel the reduced arc (and hence does not execute on the upstream node). Similarly, the $r_{i,k}$ of the downstream node is zero because it is only updated by SPLITs, but a SPLIT could not have been performed on that node because it has an incoming arc. Note also that SPLITs involving nodes with no arcs traveled by J_i (i.e., no arcs in $L_{i,k}^0$) do not update $q_{i,k}$ and $r_{i,k}$ since J_i could not have parted J_k at the split node. Consequently, $q_{i,k}$ and $r_{i,k}$ are *updated* only by PIPEs and SPLITs applied to nodes with an arc in $L_{i,k}^0$. Hence, only PIPEs and SPLITs involving arcs traveled by *both* jobs (i.e., in $Seg_{i,k}$) need to be considered.

The above observation makes the proof simple. Consider segment $s \in Seg_{i,k}$. Let E_s be the last node of this segment. Initially, each node $j \in s$ had $q_{i,k}^j = C_{i,j}$. To reduce each arc in that segment, a PIPE must have been performed producing $q_{i,k} = \max C_{i,j}$ over PIPE operands. Any SPLIT performed on nodes $j \in s$, other than the last node E_s did not affect $q_{i,k}$ and $r_{i,k}$ variables per SPLIT definition, since J_k and J_i did not part ways at j. At node E_s, J_k and J_i did part ways, but a SPLIT could not have been performed until the entire segment s was reduced to one node (because SPLIT cannot be performed for nodes with incoming arcs). In that node, $q_{i,k} = C_{i,max}^s$ since, by then, the max operator (the PIPE) will have been applied over all nodes in the segment. The SPLIT would then add $C_{i,max}^s$ to $r_{i,k}$. As we noted above, subsequent PIPEs propagate $r_{i,k}$ to the result node. When all segments have been reduced, $r_{i,k}$ will have been updated by one SPLIT at the end of each segment, each adding $C_{i,max}^s$ of one segment, resulting in $\sum C_{i,max}^s$ over all sub-jobs J_i^s, or $r_{i,k}^M = r_{i,k}^*$. (Observe that, if the end node of J_k and J_i is the same, there would be no SPLIT for the last segment and its max-term would still be stored in $q_{i,k}$, which is why we need to manually add $q_{i,k}^M$ and $r_{i,k}^M$ at the end, essentially compensating for the missing SPLIT.)

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

4. Handling Non-Acyclic Task Graphs

In this section, we describe how the system model can be extended to handle non-acyclic task graphs (i.e., those where the set of all paths of tasks taken together may contain cycles). The basic idea is to modify the non-acyclic system into an acyclic system, in a manner that does not improve the delay and schedulability of jobs. We define a third operator, called CUT.

Definition 3: CUT Operator. When the directed resource graph contains a cycle, 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 CUT operation only relaxes constraints on the arrival times of jobs, allowing jobs to arrive in a manner that can cause worse delay than when the constraint was present (an adversary has greater freedom in choosing the arrival times of jobs so as to create a worst-case delay). This decreases schedulability of the resulting task set. Hence, if the CUT set is schedulable so is the original set. The CUT operation can be repeatedly performed until all cycles have been broken. The PIPE and SPLIT operations can then be used. After obtaining the final operand matrices, to determine the delay (and schedulability) of a job J_i that has been cut into two or more sub-jobs, one needs to simply consider the cumulative delay of all the sub-jobs of job J_i. It must be noted that, the pessimism in schedulability analysis can be reduced by intelligently choosing the jobs to cut, and the stages at which to perform the cut. Determining rules or guidelines to optimize the cut operations can be an interesting future direction.

5. Evaluation

In this section, we evaluate the accuracy and tightness of the delay composition algebra in estimating the end-to-end delay and schedulability of jobs. A custom-built simulator that models periodic tasks executing in a distributed system is used. An admission controller based on the delay composition algebra is used to guarantee the deadlines of tasks in the system. When a periodic task enters the system, it is admitted if the set of all tasks is found to be schedulable, and is dropped otherwise. The analysis is meant as a design-time capacity-planning tool and hence the need for global knowledge by the admission controller is not a problem.

We consider two measures of performance. First, we estimate the average ratio of the end-to-end delay of tasks to their computed worst-case end-to-end delay bound. This metric shows how pessimistic the theoretically computed worst-case is (as per each approach) compared to the average case. Second, we consider the average per-stage utilization of tasks admitted into the system and is a measure of the throughput of the system. Utilization of a resource is defined as the fraction of time the resource is busy servicing a task. We compare our analysis using the delay composition algebra with holistic analysis [14] and network calculus [3, 4], under both preemptive and non-preemptive scheduling. We use the result from [10], for holistic analysis under non-preemptive scheduling. We build an admission controller for each analysis technique (delay composition algebra, holistic analysis, and network calculus) and compare the conservatism of the various analyses with respect to admission control. While extensions (such as [12]) have been proposed, we use holistic analysis as a basis for comparison as these extensions are targeted to handle invocation offsets (which we do not consider in this paper), and they have similar performance degradation as holistic analysis with increasing system scale.

The scheduling policy was assumed to be deadline monotonic scheduling. Tasks that enter the system, request processing at a sequence of stages, with each stage having a probability NP (for Node Probability) of being part of the task's route (default is 0.8). End-to-end deadlines (equal to the periods, unless explicitly specified otherwise) of tasks are chosen as $10^x a$ simulation seconds, where x is a uniformly varying real value between 0 and DR (for deadline ratio parameter), and $a = 500 * N$, where N is the number of stages in the task's route. This enables the ratio of the longest to the shortest task deadline to be as large as 10^{DR}. The default value for DR is 2.0. We define the task resolution parameter T, as the ratio of the total computation time of a task over all the stages on which it executes to its end-to-end deadline. The stage execution times for each task is calculated based on a uniform distribution with mean $\frac{DT}{N}$, where D is the end-to-end deadline and T is the task resolution. The default value for T is $1:20$. The stage execution times of tasks were allowed to vary up to 10% on either side of the mean. The default schedulability test used on the composed uniprocessor is the response-time analysis technique presented in [1].

Each point in the figures below represents average values obtained from 100 executions of the simulator, with each execution running for 80000 task invocations. When comparing different admission controllers, each admission controller was allowed to execute on the same 100 task sets. The 95% confidence interval for all the values presented in this section is within 1% of the mean value, which is not plotted for the sake of legibility.

First, we ascertain that the performance does not significantly drop with increasing system size. We measured the average ratio of end-to-end delay of jobs to the calculated upper bound on the worst-case delay, as a function of system size. The results are shown in Figure 4.

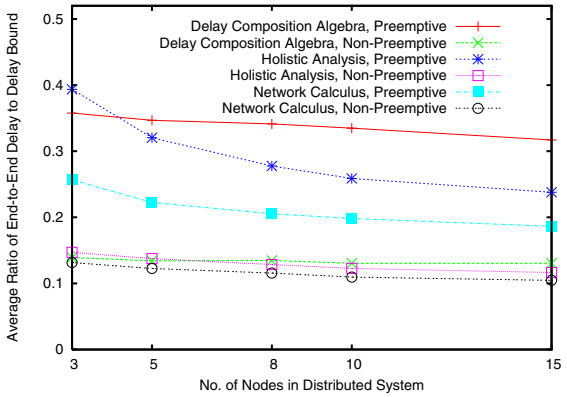

Figure 4. Comparison of average ratio of end-to-end delay to estimated delay bound for different number of nodes in the system

For the delay composition algebra, under both preemptive and non-preemptive scheduling, the ratio remains nearly the same regardless of system size, showing that the pessimism in analysis does not increase with system scale. However, holistic analysis tends to be increasingly pessimistic with system scale, and the ratio drops with increasing number of nodes in the system. The ratio is lower for non-preemptive scheduling, as there are several low priority jobs that finish well before their worst-case delay estimate as they are not

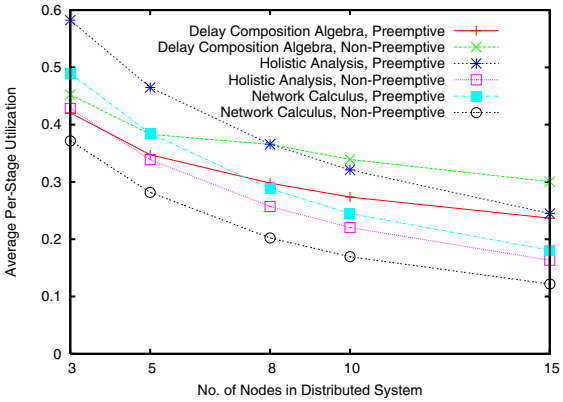

Figure 5. Comparison of average per-stage utilization for different number of nodes in the system

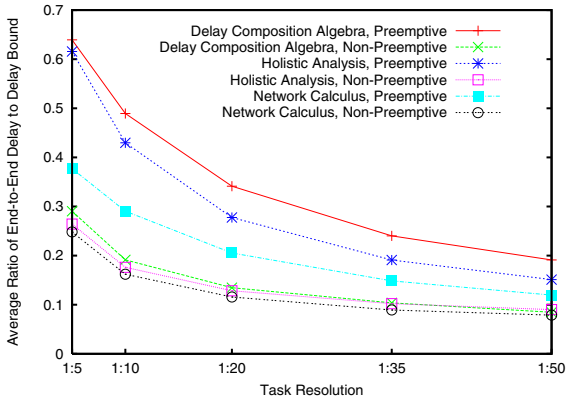

Figure 6. Comparison of average ratio of end-to-end delay to estimated delay bound for different task resolution values

preempted by higher priority jobs and therefore encounter only a smaller fraction of all higher priority jobs during their execution (on an average) than under preemptive scheduling.

For the same experiment, Figure 5 plots the average per-stage utilization of admitted tasks. Note that, the drop in average utilization is faster for holistic analysis and network calculus than for our algebraic analysis with increasing system size. Holistic analysis consistently outperforms network calculus for all system sizes.

We next varied the size of jobs by adjusting the task resolution parameter T. A large value for T (e.g., 1:5) denotes a system with a small number of large tasks, and a small value of T (e.g., 1:50) denotes a large number of small tasks. We measured the ratio of the end-to-end delay to the delay bound for the three analysis techniques under both preemptive and non-preemptive scheduling, and the results are shown in Figure 6. Delay composition algebra tends to be the least pessimistic under preemptive as well as non-preemptive scheduling. As the number of tasks in the system increases (as T decreases), jobs encounter a smaller fraction of higher priority jobs, and therefore the average end-to-end delay significantly differs from the worst-case delay. Under non-preemptive scheduling, when task sizes are large (T is large) the blocking penalty for higher priority jobs is also high, although on an average jobs are not blocked for the estimated worst-case period. This causes the ratio under non-preemptive scheduling to be lower than under preemptive scheduling.

Due to paucity of space, we do not show results from varying other parameters such as the ratio of task deadlines to periods, which further confirm that using delay composition algebra leads to less pessimistic schedulability analysis of tasks in distributed systems.

6. Related Work

The technique of reducing large complex systems into smaller and simpler components that are easier to analyze have been used in contexts outside real-time systems. In control theory, for example, there exist rules for transforming complex block diagrams into an equivalent single block that can be analyzed for stability, convergence, and performance properties. In circuit theory, laws such as the Kirchoff's laws, permit the reduction of a circuit into a single equivalent source and impedance. Such transformation and reduction techniques help to largely reduce the complexity of analysis. In this paper, we define primitives to enable distributed systems serving delay constrained tasks to be reduced to equivalent uniprocessors, which can then be analyzed using traditional uniprocessor analysis techniques.

The schedulability of real-time tasks in distributed systems has been analyzed using several techniques with different time complexities. The compositional framework presented in this paper is unique in the sense that it provides a systematic way to reduce complex distributed systems into equivalent single stage systems, thereby largely reducing the complexity of analysis.

There has been a lot of work in the context of job-fair scheduling, where the objective is to obtain a feasible schedule of executing the tasks on a pipelined distributed system. A representative example is the work in [2], where polynomial-time algorithms were developed for certain special cases and heuristic algorithms were proposed for the general case. Algorithms have been proposed to statically schedule precedence constrained tasks in distributed systems [15, 5]. These algorithms construct a schedule of length equal to the least common multiple of the task periods, that would precisely define the time intervals of execution of each job. Offline schedulability tests have been proposed that divide the end-to-end deadline of tasks into per-stage deadlines. Uniprocessor schedulability tests are then used to analyze if each stage is schedulable. For instance, offset-based response time analysis techniques for EDF were proposed in [11, 13].

Holistic schedulability analysis for distributed systems [14], estimates the worst case response time of tasks by assuming that the worst-case delay at a stage to be the jit-

ter for the next stage. This technique requires global knowledge of all task routes and computation times in order to estimate the end-to-end delay of any given task. In contrast, to analyze the delay and schedulability of a particular task, the algebra presented in this paper only requires knowledge of tasks along its path. An extension to holistic analysis to handle resource blocking under non-preemptive scheduling was presented in [10]. Network calculus has been derived in [3, 4] to analyze delay and burstiness of flows in networks of resources. While being more general and more widely applicable than delay composition algebra, it tends to be very pessimistic especially for large systems. In [6], a utilization-based schedulability test for analyzing aperiodic tasks under fixed priority scheduling was proposed. While this test is very simple to apply, it nevertheless does not account for the parallelism in the execution of different pipeline stages.

In earlier publications [7, 8], we derived a delay composition theorem that bounds the worst-case end-to-end delay of jobs in pipelined systems under preemptive and non-preemptive scheduling. We extended them to directed acyclic graphs, and also to partitioned resources (e.g., TDMA scheduling) in [9]. While these results were from the perspective of a single job in the system, in this paper, we take a system-wide view and present an algebra for composing distributed system components. Further, we present ways to extend the results to non-acyclic systems.

7. Conclusion and Future Work

In this paper, we present an algebra, which we call delay composition algebra, for composing together resource stages in real-time distributed systems, under both preemptive and non-preemptive scheduling. We define operators to compose resource stages based on the traversal pattern of jobs in the system. By successively applying the operators of the algebra on operands that represent tasks in resource stages, the distributed system can be reduced to an equivalent single stage that can be analyzed to infer end-to-end delay and schedulability of jobs in the distributed system. We show using simulation studies that the proposed algebraic framework is less pessimistic with increasing system scale compared to traditional approaches.

While the algebra developed in this paper can handle a general class of distributed systems, it has several limitations that motivate future work. The operators defined in this paper can be improved to reduce the pessimism of the analysis. For instance, the reduction of the multi-stage system to an equivalent single stage disregards information regarding the particular set of stages on which each job executed. The uniprocessor analysis can be improved if the operators make this information available, perhaps at the cost of simplicity. While this paper describes a technique to handle non-acyclic systems, the solution involves converting the non-acyclic system into an acyclic system, which can then be analyzed. A set of operators that handle non-acyclic systems naturally without having to convert the system into an acyclic system, can lead to improved analysis. The theory also needs to be extended to handle partitioned resources (e.g., TDMA scheduling), systems where each job need not have the same priority on every resource, and systems where jobs may simultaneously require more than one resource.

References

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

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

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

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

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

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

[7] P. Jayachandran and T. Abdelzaher. A delay composition theorem for real-time pipelines. In *ECRTS*, pages 29–38, July 2007.

[8] P. Jayachandran and T. Abdelzaher. Delay composition in preemptive and non-preemptive real-time pipelines. *Invited to Real-Time Systems Journal: Special Issue on ECRTS'07 (to appear)*, 2008.

[9] P. Jayachandran and T. Abdelzaher. Transforming acyclic distributed systems into equivalent uniprocessors under preemptive and non-preemptive scheduling. In *ECRTS*, pages 233–242, July 2008.

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

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

[12] J. Palencia and M. G. Harbour. Schedulability analysis for tasks with static and dynamic offsets. In *IEEE Real-Time Systems Symposium*, pages 26–37, December 1998.

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

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

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