Flow-based Mode Changes: Towards Virtual Uniprocessor Models for Efficient Reduction-based Schedulability Analysis of Distributed Systems*

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

Abstract

This paper is the first to consider new uniprocessor task models motivated by the needs of reduction-based schedulability analysis techniques for distributed systems. Reduction-based analysis is a recent category of distributed system schedulability analysis techniques that reduces distributed real-time workloads to equivalent virtual uniprocessor ones for purposes of analysis using classical uniprocessor techniques. The approach motivates research on uniprocessor task models that better match the peculiarities of task loads reduced from distributed systems. We show that previous reduction-based schedulability analysis techniques suffer from pessimism that results from mismatches between uniprocessor analysis assumptions and characteristics of workloads reduced from distributed systems. To address the problem, we introduce flow-based mode changes, a uniprocessor load model tuned to the novel constraints of workloads reduced from distributed system tasks. Reducing distributed workload to this model, our simulation studies suggest that the resulting schedulability analysis is able to admit over 25% more utilization than other existing techniques, while still guaranteeing that all end-to-end deadlines of tasks are met.

1. Introduction

In this paper, we develop a new task model for uniprocessors, motivated by the needs of reduction-based schedulability analysis techniques for distributed real-time systems. Reduction based approaches to schedulability [6, 7, 8] transform distributed system workloads into equivalent uniprocessor workloads that can be analyzed using techniques borrowed from uniprocessor literature. The resulting uniprocessor workloads are subject to additional constraints not previously considered in that literature. The current practice has been to ignore these constraints (on worst-case load), resulting in needlessly pessimistic worst cases, and hence in pessimistic schedulability estimates for workloads reduced from distributed systems. The problem motivates new uniprocessor workload models that serve the needs of reduction-based schedulability analysis literature.

The fundamental idea of workload transformation in reduction-based schedulability analysis is to show that when two periodic distributed tasks execute together on a sequence of machines (called stages) in a distributed system, each invocation of the higher-priority task delays an invocation of the lower-priority task by a bounded total amount in the entire system. This bounded amount is computed and becomes the transformed execution time of an equivalent periodic higher-priority task on a virtual uniprocessor. Analyzing the schedulability of the lower priority task subject to all such transformed higher-priority uniprocessor periodic tasks then determines the schedulability of the former in the original distributed system.

The source of pessimism arises due to the fact that once all tasks have been reduced to a single periodic uniprocessor task set, uniprocessor analysis assumes that these tasks execute together “all the time”, whereas in fact they may share a machine only for part of their execution in the original distributed system. Hence, the number of interfering invocations of higher-priority tasks may be overestimated.

We address the pessimism in current reduction-based schedulability analysis techniques by introducing flow-based mode changes, a novel model for uniprocessor workloads featuring mode changes that mimic what happens when a distributed task moves from one processor to another in a distributed system. A key distinction of our model as compared to the classical literature on mode changes, such as [16, 18, 15, 14], is that we do not precisely know when the mode changes will occur, as we do not know when exactly the task changes machines. However, we have constraints on mode change timing that stem from bounds on task delays on different machines. Therefore, the problem is one of estimating the response time of a task for the worst-case scenario of mode changes subject to new mode change constraints.

The paper presents an iterative solution to solve the above problem, providing significantly tighter estimates on the number of higher priority task invocations that delay the task under consideration. The solution converges to a delay bound that never underestimates the worst-case delay of the corresponding task in the distributed system. Our simulations demonstrate that the presented analysis provides an improvement in performance of over 25% compared to existing techniques, in terms of admissible utilization.

The rest of the paper is organized as follows. In Section 2, we describe the new multi-modal uniprocessor system model proposed in this paper. We present an iterative solution to determine the response time of a task under consideration in such a system in Section 3. In Section 4, we show the reduction of a distributed system to such a multi-modal unipro-
cessor system for the purpose of schedulability analysis. We also present an example to illustrate the advantage of the proposed solution over previous reduction based analysis techniques for distributed systems. We evaluate the technique through simulation in Section 5. We review related work in Section 6 and conclude the paper in Section 7.

2. Multi-Modal Uniprocessor System Model

In this section, we present the new multi-modal uniprocessor system model of interest in this paper. This model is motivated by the needs of reduction-based schedulability analysis in distributed systems. It is thus important to first highlight the relation between distributed task execution models and the model below.

Reduction-based schedulability analysis addresses scenarios where tasks execute on a sequence of machines in a distributed system and must each finish within its end-to-end deadline. Consider a distributed system, running fixed priority scheduling, where tasks $T_1, T_2, \ldots, T_m$, are executed, indexed in decreasing priority order. Task $T_i$ executes on a sequence of $n_i$ machines. It therefore comprises of a sequence of $n_i$ jobs $T_{i,1}, T_{i,2}, \ldots, T_{i,n_i}$, each running on the corresponding machine. Job $T_{i,j+1}$ becomes ready to execute as soon as $T_{i,j}$ completes execution, at which point task $T_i$ is said to have switched to the next machine on its path.

The model fits a pipelined execution scenario, where a task is broken up into a number of sequential stages that must execute in some given order.

It is now possible to plot the execution of task $T_i$ from its entry to the system to its exit from the system on a single time line. This time line will comprise one busy period (i.e., a period with no idle time) composed of intervals when the task was executing or preempted by others on some machine as well as intervals where it was executing. The finish time of each job $T_{i,j}$ in that time line corresponds to a time when $T_i$ switches machines and starts competing with a different task set. Assume we are able to accurately bound the delay that each invocation of a higher priority task $T_{j}, j < i$, imposes on $T_i$ in that time line (which is what reduction-based schedulability analysis literature does). One will then need only to bound the maximum number of invocations of each $T_{j}$ that may delay $T_i$ in order to bound $T_i$’s end-to-end response time. This is not straightforward, however, because $T_i$ competes with potentially different subsets of higher priority tasks on each machine, and the exact times it changes machines are not known accurately as they depend on the relative timing of task arrivals. To bound the delay of $T_i$, schedulability analysis of this time line can then benefit from a uniprocessor model where “mode changes” occur at $T_{i,j}$’s completion times.

The objective of that model is to come up with the worst-case timing for “more changes” that maximize $T_i$’s response time (e.g., letting $T_i$ spend longer on machines with heavier load).

Note that, the above maximization problem does not simply amount to the sum of $T_i$’s worst-case response times on each machine. This would be needlessly pessimistic. For example, if $T_i$ and the higher-priority tasks followed the same path, arriving at the first machine together (a worst-case arrival scenario on the first machine), then they will arrive to the next machine staggered by their execution times on the first (which is not a worst-case arrival scenario). Below, we present a multi-modal uniprocessor model that achieves a much tighter response-time bound, and then describe how it can be used for analysis of distributed workloads.

Consider a uniprocessor that uses fixed priority preemptive scheduling. We consider a set of $m$ tasks $T_1^*, T_2^*, \ldots, T_m^*$, in decreasing priority order. Task $T_m^*$, the lowest priority task whose worst-case response time we wish to estimate, comprises of a sequence of $n$ jobs $T_{m,1}^*, T_{m,2}^*, \ldots, T_{m,n}^*$, with computation times $C_{m,1}^*, C_{m,2}^*, \ldots, C_{m,n}^*$, respectively. The jobs are such that $T_{m,j+1}^*$ is ready to execute as soon as $T_{m,j}^*$ completes execution, for $1 \leq j \leq n - 1$. Job $T_{m,1}^*$ is ready to execute at time zero. The time instant of completion of each of the jobs $T_{m,j}^*$ denotes a mode change in the system, where one of the other $m - 1$ tasks may arrive or leave the system. Tasks $T_i^*$, $i \leq m - 1$ are periodic tasks with period $P_i$ and computation time $C_i^*$.

Each task $T_i^*$ arrives at the system at time zero, or during one of the mode changes in the system (time instants of completion of $T_{m,j}$, for some $j < n$), and leaves the system at one of the mode changes or when $T_{m,n}$ completes execution. Thus, each periodic task executes during some consecutive subset of modes in the system and does not undergo any change within this subset of modes until it leaves the system. The subset of modes in which a task executes is assumed to be known. Hence, during each mode $mode_j$, $j \leq n$ of execution, some pre-defined subset of periodic tasks $L_j$ is present in the system.

We assume that task preemptions and mode changes are instantaneous. We also assume that the cumulative utilization of all the tasks executing during any mode is at most 100%. The objective is to estimate a worst-case bound on the response time of $T_{m,n}^*$ starting from time zero, over all possible scenarios of mode changes. Note that, unlike traditional multi-modal analysis, we are not interested in the schedulability of all tasks in the system, but are interested only in that of $T_{m,n}^*$. As each task in the distributed system could have a different worst-case scenario, the analysis is conducted one task at a time and a different multi-modal uniprocessor system is constructed and analyzed each time. We later show in Section 4, how a distributed task set can be reduced to such a multi-modal uniprocessor, and how the computed bound for $T_{m,n}^*$ also bounds the end-to-end delay of the corresponding task in the distributed system.

3. Schedulability Analysis

In Section 3.1, we present an algorithm to determine the worst-case response time of $T_{m,n}^*$ in the multi-modal uniprocessor system (we use $T_{m,n}^*$ interchangeably to denote the entire task invocation or its last stage when analyzing completion time). We illustrate the algorithm using an example in Section 3.2. We comment on the time complexity of the algorithm in Section 3.3.
3.1 Algorithm Description

Consider Figure 1 that demonstrates the execution of task $T_m$, the (yet to be determined) instances of mode changes, and the arrival and departure of higher priority tasks. The completion of the sub-task $T_{m,j}$ denotes the completion of $mode_j$, for each $j$. Let the set of tasks that execute in $mode_j$ be denoted by $L_j$.

![Diagram showing mode changes and task execution](image)

**Figure 1:** Example demonstrating the instants of mode changes and the arrival and departure of higher priority tasks

Let $RT(s, s + q)$, $s \geq 0, q \geq 1$, denote the maximum possible duration between the completion of $mode_s$ and the completion of $mode_{s+q}$. For notational simplicity, we consider time zero (the arrival of task $T_m$) to be the “completion” of a fictional $mode_0$. Therefore, we are interested in computing $RT(0, n)$, which denotes the worst-case response time of $T_{m,n}$.

Given the set of tasks $L_s$ that execute at each mode $mode_s$, we can apply response time analysis [1] to compute the maximum response time, $RT(s, s + 1)$ for each mode taken independently. Adding up these worst-case single-mode response times for $s = 0, \ldots, n - 1$ would give us an upper bound on $RT(0, n)$. However, such an upper bound will be unduly pessimistic. To appreciate the reason for pessimism, consider a task $T^*_m$ that executes in modes $mode_s$ and $mode_{s+1}$ (e.g., task $T^*_3$ in Figure 1 that executes in $mode_3$ and $mode_4$). Let the period of $T^*_m$ be larger than the total length of the two modes combined (remember that the end of each $mode_s$ is defined as the instant when $T_{m,s}$ ends, and hence is not necessarily aligned with periods of other tasks). Since there can only be one invocation of $T^*_m$ in each period, this invocation will execute either in $mode_s$ or $mode_{s+1}$ but not both. The worst-case response time computed for each mode separately will have to account for this invocation of $T^*_m$. However, adding these worst-case response times will erroneously double-count this invocation. Hence, in general:

$$RT(s, s + 2) \leq RT(s, s + 1) + RT(s + 1, s + 2)$$

In order to compute a less pessimistic estimate, $RT(0, n)$, of the worst-case completion time of $T_{m,n}$, we cast the problem as one of dynamic programming, as shown in Table 1. In this table, the first column computes the worst-case single-mode durations, $RT(s, s + 1)$. The $q^{th}$ column computes the worst-case durations of all possible sequences of $q$ consecutive modes, $RT(s, s + q)$, using data from the previous columns, while avoiding double-counting as we shall explain shortly. Observe that there are $n - q + 1$ rows in column $q$.

The last column yields $RT(0, n)$, which is the solution to our problem.

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>q</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>RT(0,1)</td>
<td>RT(0,2)</td>
<td>RT(0,q)</td>
<td>RT(0,n)</td>
</tr>
<tr>
<td>RT(1,2)</td>
<td>RT(1,3)</td>
<td>RT(1,q+1)</td>
<td></td>
</tr>
<tr>
<td>RT(2,3)</td>
<td>RT(2,4)</td>
<td>RT(2,q+2)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RT(n-2,n-1)</td>
<td>RT(n-2,n)</td>
<td>RT(n-q,n)</td>
<td></td>
</tr>
<tr>
<td>RT(n-1,n)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Table 1:** Table illustrating the values computed using dynamic programming

Trivially, the first column can be computed from response time analysis [1], as follows:

$$RT(s, s + 1) = C_{m,s+1} + \sum_{T^*_i \in L_{s+1}} \left[ \frac{RT(s, s + 1)}{P_i} \right] C^*_i$$

The above equation assumes that all higher-priority tasks are strictly periodic. Hence, in an interval of length $t$, there can be at most $\lceil t / P_i \rceil$ invocations of task $T^*_i$. According to reduction-based schedulability analysis [6, 7, 8], the uniprocessor tasks that result from reduction of distributed systems are strictly periodic if they arrive strictly periodically to the first shared stage with $T_m$ (whose schedulability is being analyzed). In general, task $T^*_i$ may have jitter $J_i$ defined as the worst-case delay in its arrival time past the nominal beginning of its period. Thus, during a time interval, $t$, there may be as many as $\lceil (t + J_i) / P_i \rceil$ invocations, as shown in Figure 2.

![Diagram showing system with and without jitter](image)

**Figure 2:** (a) System without jitter, (b) System with jitter

The maximum response time (i.e., the content of the first column of Table 1) is therefore more generally written as:

$$RT(s, s + 1) = C_{m,s+1} + \sum_{T^*_i \in L_{s+1}} \left[ \frac{RT(s, s + 1) + J_i}{P_i} \right] C^*_i$$

Each subsequent column $q$ in the dynamic programming ta-
where \( in_t \) is the larger of \( s \) and the mode after which task \( T^*_i \) enters the system, and \( out_t \) is the lower of \( s + q \) and the mode after which \( T^*_i \) exits. In other words, for each task \( T^*_i \), we compute the maximum number of invocations that can potentially delay \( T^*_m \) between (the ends of) mode \( s \) and mode \( s + q \). Note that, since \( in_t \geq s \) and \( out_t \leq s + q \), as defined above, all terms \( RT(in_t, out_t) \) will have been computed from previous iterations, except the term \( RT(s, s + q) \), leaving the above equation a function of \( RT(s, s + q) \) on both sides, which can be solved recursively. When the dynamic programming algorithm terminates (at \( q = n \)), \( RT(0, n) \) is returned as the final answer.

**Algorithm**

1. For \( s = 0 \) to \( n - 1 \)
   1.1 Set \( RT(s, s + 1)^{(0)} = C^*_m, s + 1, k = 0 \)
   1.2 Repeat: Increment \( k \)
      \[
      RT(s, s + 1)^{(k)} = C^*_m, s + 1 + \sum_{T^*_i \leq L, j\geq 1} \left[ \frac{RT(s, s + 1)^{(k)} + J_i}{P_i} \right] C^*_i
      \]  
      Until \( RT(s, s + 1)^{(k)} = RT(s, s + 1)^{(k-1)} \)
   1.3 Let \( RT(s, s + 1) = RT(s, s + 1)^{(k)} \)
2. For \( q = 2 \) to \( n \)
   2.1 For \( s = 0 \) to \( n - q \)
   2.2 Set \( RT(s, s + q)^{(0)} = \sum_{s < j \leq s + q} C^*_m, j, k = 0 \)
   2.3 Repeat: Increment \( k \)
      \[
      RT(s, s + q)^{(k)} = \sum_{s < j \leq s + q} C^*_m, j + \sum_{T^*_i \geq L, j\geq 1} \left[ \frac{RT(in_t, out_t) + J_i}{P_i} \right] C^*_i,
      \]  
      where \( in_t = \max(s, arr_t) \), and \( out_t = \min(s + q, leave_t) \)
      Until \( RT(s, s + q)^{(k)} = RT(s, s + q)^{(k-1)} \)
   2.4 Let \( RT(s, s + q) = RT(s, s + q)^{(k)} \)
End for
3. Return \( RT(0, n) \)

**Figure 3:** Algorithm for analysis of a uniprocessor with flow-based mode changes

We summarize the algorithm in Figure 3. Let \( arr_t \) denote the mode after which \( T^*_i \) enters the system and executes together with \( T^*_m \), and \( leave_t \) denote the mode after which \( T^*_i \) leaves the system. In step 1 of the algorithm, we consider each mode independently and apply response time analysis to determine the maximum duration of each mode. In step 2, we consider \( q \) consecutive modes taken together, for increasing values of \( q \), and determine the maximum duration of \( RT(s, s + q) \) for all \( s \leq n - q \), given the values of \( RT(in_t, out_t) \), for all \( out_t - in_t < q \). Finally, the value \( RT(0, n) \) computes the worst-case response time of \( T^*_m \).

The correctness of the algorithm follows trivially from the fact that Equation (1) and Equation (2) never underestimate the number of invocations of higher-priority tasks that preempt \( T^*_m \) in the time intervals under consideration, and hence never underestimate the computed time intervals, including the solution \( RT(0, n) \).

### 3.2 Example to Illustrate the Algorithm

We now illustrate the above algorithm using a simple example. Consider a uniprocessor system with flow-based mode changes. Let the system have 5 modes and 7 periodic tasks \( T_1, T_2, \ldots, T_7 \), in decreasing priority order (the corresponding distributed system is shown in Figure 4). We are interested in computing the response time of the lowest priority task \( T_7 \). Tasks \( T_0 \) and \( T_7 \) operate during all modes. Tasks \( T_1, T_2, \ldots, T_5 \), each execute at one of the 5 modes. In particular, \( mode_j \), \( 1 \leq j \leq 5 \), has tasks \( T_j, T_{6-j} \), and \( T_7 \). For simplicity, let us assume that all tasks are strictly periodic and have no input jitter. Let all the higher priority tasks, \( T_1, T_2, \ldots, T_0 \), have a unit execution time per period, and let \( T_7 \) have an execution time of 0.5 in each mode. Let the period (equal to the deadline) of \( T_0 \) be 100 units, \( T_7 \) be 200 units, and \( T_1, T_2, \ldots, T_5 \) be 5 units. The above task set is summarized in the table below.

<table>
<thead>
<tr>
<th>Task</th>
<th>( C_i )</th>
<th>( P_i )</th>
<th>Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T_1 )</td>
<td>1</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>( T_2 )</td>
<td>1</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>( T_3 )</td>
<td>1</td>
<td>5</td>
<td>3</td>
</tr>
<tr>
<td>( T_4 )</td>
<td>1</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>( T_5 )</td>
<td>1</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>( T_6 )</td>
<td>100</td>
<td>All</td>
<td>All</td>
</tr>
<tr>
<td>( T_7 )</td>
<td>0.5/mode</td>
<td>200</td>
<td>All</td>
</tr>
</tbody>
</table>

**Table 2:** An example task set

We first compute \( RT(s, s + 1) \), for every \( s \), using response time analysis. We obtain \( RT(s, s + 1) = 2.5 \), for every \( s \), as \( 2.5 = 0.5 + \lceil 2.5/5 \rceil + \lceil 2.5/100 \rceil \). Next, we consider two
consecutive modes taken together. For every $s$:

$$RT(s, s+2) = 0.5 + 0.5 + \left\lceil \frac{2.5}{5} \right\rceil + \left\lceil \frac{2.5}{5} \right\rceil + \frac{RT(s, s+2)}{100}$$

Solving, we get $RT(s, s+2) = 4$. Similarly:

$$RT(s, s+3) = 0.5 + 0.5 + 0.5 + \left\lceil \frac{2.5}{5} \right\rceil + \left\lceil \frac{2.5}{5} \right\rceil + \frac{RT(s, s+3)}{100}$$

Solving, we get $RT(s, s+3) = 5.5$. Proceeding similarly, we obtain $RT(s, s+4) = 7$, and $RT(0, 5) = 8.5$. Therefore, the end-to-end response time of $T_i$ is computed as 8.5. In this simple example, the bound is tight. Indeed, $T_i$ has a total execution time of 2.5 over the five modes, and can be preempted by at most one invocation of each higher-priority task (of 1 unit of execution time each). This adds up to 8.5 units. In general, our bound is not tight. Pessimistic estimates are possible.

3.3 Time Complexity of the Algorithm

The time complexity of the algorithm is certainly pseudo-polynomial. For an $n$ stage pipeline, the number of $RT(in, out)$ terms that need to be computed is $n + (n-1) + (n-2) + \ldots + 1 = n(n-1)/2$, with each term requiring a recursive computation until convergence. This is the price we pay for the improved accuracy in determining the end-to-end response time of tasks. However, it must be noted that the iterative algorithm needs to be executed for only one equivalent hypothetical uniprocessor, unlike techniques such as [21, 5] that attempt to construct the entire schedule for all the stages in the distributed system, in order to determine the worst-case end-to-end response time.

4. End-to-End Delay Analysis of Distributed Tasks

In this section, we show how the end-to-end delay analysis of a distributed task can be reduced to that of analyzing a hypothetical multi-modal uniprocessor, modeled in Section 2. First, in Section 4.1, we briefly describe the distributed task model considered in this paper. We present the transformation and show how it works in Section 4.2.

4.1 Distributed System Model

The distributed system model considered in this paper is similar to the model assumed in [8], and we describe it briefly here for convenience. The system running fixed priority preemptive scheduling, consists of $N$ resource nodes $S_1, S_2, \ldots, S_N$, and a set of $M$ periodic tasks, $T_1, T_2, \ldots, T_M$, ordered by decreasing priority. Each task requires service at some sequence of resources and must complete execution on all resources before a pre-specified end-to-end deadline. Task paths can be cyclic, that is, a task can revisit a resource multiple times before leaving the system.

Let $T_M$ be the task whose end-to-end delay is to be estimated. Note that, a task $T_i$ can delay $T_M$ only along execution nodes it shares in common with $T_M$. We define a task segment $T_i^x$ as $T_i$’s execution on a sequence of consecutive nodes along its path that is also traversed by $T_M$ either in the same order or exactly in reverse order. We ignore the precedence constraints between different segments of each higher priority task $T_i$, and consider each segment as an independent task. This procedure is illustrated in Figure 5. As explained in [8], note that this does not decrease the end-to-end delay of $T_M$ as we only remove certain precedence constraints, thereby increasing the set of possible arrival patterns of higher priority tasks to stages. By considering this larger set of arrival patterns of higher priority tasks and computing $T_M$’s delay bound, we can only obtain a result that is larger than one obtained by considering all the precedence constraints (thus erring on the safe side).

Let $C_{i,j}$ denote the worst-case execution time of an invocation of $T_i$ on stage $j$ (referred to simply as execution time or stage execution time), and let $D_i$ and $P_i$ denote the end-to-end deadline and invocation period of $T_i$, respectively. Let $C_{i,\text{max}}$ denote the maximum computation time of $T_i$ across all the stages on which it executes, and let $N_{\text{node},j,\text{max}}$ denote the maximum computation time over all tasks that execute on stage $j$. We assume that $D_i \leq P_i$, for all $i$.

4.2 Distributed System Transformation to an Equivalent Uniprocessor with Mode Changes

As mentioned in Section 4.1, we consider each higher priority task segment as independent. After relaxing the precedence constraints, the system can now be thought of as having an arbitrary set of $m-1$ higher priority tasks ($m \geq M$, as we are breaking each task into multiple independent task segments), each executing on a sequence of consecutive common stages with the lowest priority task $T_m$, whose worst case end-to-end delay is to be estimated. According to the
According to the delay composition theorem [8], each higher priority task segment $T^*_k$ contributes a delay of at most twice its maximum stage computation time, i.e., $2C_{i,max}$ to the end-to-end delay of $T_m$. Apart from the per-task delay, $T_m$ also experiences a delay component that is additive across the stages on which it executes. For each stage on which it executes, it experiences a delay that is bounded by the maximum computation time of any task on that stage, namely $Node_{j,max}$. For the sake of completeness, we reproduce the statement of the delay composition theorem from [8] here. Let $Seg_i$ denote the set of all task segments of task $T_i$.

**Delay Composition Theorem.** 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 $J_m$ following path $p_m$ can be composed from the execution parameters of higher priority jobs that delay or preempt it as follows:

$$\text{Delay}(J_m) \leq \sum_{i=1}^{n} \sum_{j \in Seg_i} 2C_{i,max} + \sum_{j \in p_m} Node_{j,max}$$  

(3)

Let $n$ denote the number of stages in the path of $T_m$. For simplicity, we renumber the stages on which $T_m$ executes as $S_1, S_2, \ldots, S_5$. The reduction of the distributed system to a multi-modal uniprocessor system is conducted as follows:

- Corresponding to each of the stages on which $T_m$ executes, we create a sequence of $n$ jobs $T_{m,1}, T_{m,2}, \ldots, T_{m,n}$ on the uniprocessor with the same priority as $T_m$, with computation times equal to $Node_{j,max}$, for $j \leq n-1$, and equal to $C_{m,max}$, for the $n$th stage. $T_{m,j}$ is ready to execute on the uniprocessor right when $T_{m,j-1}$ completes execution, for $j \geq 2$. $T_{m,1}$ is ready to execute at time zero.

- Corresponding to each higher priority task $T_i$ that executes between stages $S_j$ and $S_k$, we create a uniprocessor task $T^*_i$ with the same priority and period as $T_i$, and with computation time equal to $2C_{i,max}$, which is twice the maximum stage computation time of $T_i$. $T^*_i$ is ready to execute when $T_{m,j-1}$ completes execution (or at time zero if $j = 1$) and is removed from the uniprocessor system when $T^*_{m,k}$ completes execution.

Thus, time instants where $T_{m,j}, j < n$, complete execution, act as instants of mode change in the system. A set of tasks may leave the system at this instant, and a new set of tasks may enter. Let $I_j$ denote the set of higher priority tasks that execute together with $T_{m,j}$ on the uniprocessor during mode $j_i$ (these are the set of tasks that execute together with $T_m$ on stage $S_j$ in the distributed system). From when the system starts execution at time zero, we are interested in the worst case time at which $T_{m,n}$ completes execution. We presented an iterative solution that was shown to converge to an upper bound on the worst-case response time of $T_{m,n}$ in Section 3. We shall now show that this indeed provides an upper bound to the worst-case end-to-end delay of $T_m$ in the distributed system.

**4.3 An Example**

In this section, we illustrate the advantage of using the analysis presented in this paper over previous reduction based analysis techniques, using an example. In [8], by reducing the distributed system to a single static set of periodic tasks on the uniprocessor, the analysis assumed that each higher priority periodic task interferes with a lower priority task throughout its execution. However, in the original distributed system, they interfere only at stages where both tasks execute together. Thus, for the case of periodic tasks, this reduction is pessimistic as it does not take into account the set of stages where a task $T_i$ can delay $T_m$. Therefore, by modeling stage transitions of a task in the distributed system as mode changes in the equivalent uniprocessor, we enhance the system model with information regarding when each higher priority task $T_i$ interferes with a lower priority task $T_m$ under consideration.

![Figure 6: Example system](image-url)
Computation time of 5 units, and $T^*_1$ and $T^*_2$ have a computation time of 2 units. $T^*_1$ has a period of 5 units, and all invocations of $T^*_1$ that arrive before $T^*_2$ completes execution of 5 time units, delay $T^*_2$ in the uniprocessor.

As $T_1$ and $T_3$ are the only tasks that execute on $S_1$ and $S_2$ in the distributed system, $T_3$ will complete execution on stage $S_2$ no later than 4 time units (2 for executing $T_1$ and 2 for executing $T_3$) after $T_3$ arrives at stage $S_1$. Thus, at most one invocation of $T_1$ may delay $T_3$ in the distributed system. This is shown in Figure 7(a). In contrast, in the hypothetical uniprocessor, in fact, 3 invocations of $T^*_1$ are accounted as delaying $T^*_3$, thus significantly overestimating the worst-case delay (see Figure 7(b)). Using the analysis presented in this paper, $T^*_1$ leaves the system after mode. The worst case response times of mode and mode taken independently, namely $RT(0, 1)$ and $RT(1, 2)$ are first calculated as 3 time units each. Next, $RT(0, 2)$ is calculated as $1 + 1 + 2[RT(0, 2)/5] = 4$ time units. By accurately estimating the maximum duration for which $T^*_1$ executes together with $T^*_3$, only one invocation of $T^*_1$ is accounted for as delaying $T^*_3$.

Figure 7: (a) Worst-case execution on $S_1$ and $S_2$ in distributed system, (b) 3 invocations of $T^*_1$ delay $T^*_3$ on the uniprocessor.

5. Evaluation

In this section, we evaluate the performance of the end-to-end delay analysis technique for distributed systems proposed in this paper. We compare with three other existing analysis techniques, namely, holistic analysis [19], network calculus [3, 4], and the meta-schedulability test presented in [8]. While there have been extensions proposed to both holistic analysis and network calculus, their fundamental principle is the same, and they suffer from similar performance degradation for large systems. The response time analysis technique presented in [1], is used as the uniprocessor test for the meta-schedulability test. We use a custom-built simulator with an admission controller for each of the four schedulability analysis techniques. Each new task is tentatively added to the set of all tasks, and is admitted if the admission controller can guarantee that all end-to-end deadlines will be met (this is repeated with each of the four admission controllers). We consider the average per-stage utilization of the admitted tasks as the metric of interest.

We consider an acyclic distributed system consisting of $N$ resource stages. As we are interested in the performance of large systems, the default value of $N$ is assumed to be 20. A set of periodic tasks require execution on a sequence of resource stages in the system, and needs to complete execution on all its stages within a pre-specified end-to-end deadline. While the analysis presented in this paper applies to any fixed priority scheduling algorithm, we assume that deadline monotonic scheduling is used at each stage. Task routes are chosen such that each node has a probability of $node.prob$ of being part of the route. The default value of $node.prob$ is 0.8. A task’s path is simply the sequence of selected nodes taken in increasing order.

End-to-end deadlines of tasks (equal to the task periods, unless otherwise mentioned) are chosen as $500 \times N \times 10^2$, where $x$ is a uniformly distributed random variable that takes values between 0 and $DR$ (for deadline ratio parameter). This allows the largest to shortest end-to-end deadlines of tasks to vary by as much as $10^{DR}$. The default value of $DR$ is 2.0. The stage computation time for each task is chosen based on a uniform distribution with mean $DT/N$, where $D$ is the end-to-end deadline of the task, $N$ is the number of resources in the system, and $T$ is called the task resolution parameter. The task resolution parameter is the ratio of the total computation time of the task over all the stages on which it executes to its end-to-end deadline. We used a value of $T = 1/20$. The stage computation times of each task is allowed to vary by up to 10% on either side of the mean. Such a choice of computation times in proportion to their deadlines ensures that tasks with similar priority values have comparable computation times. When the $DR$ value is low, then the computation times of all the tasks are comparable. When the $DR$ value is high, then the computation times of tasks are also widely varying. As in most applications, tasks with larger computation times also have longer deadlines. In this evaluation, we assume that the initial jitter for all tasks is taken to be zero.

We ignore preemption overheads and task preemptions are assumed to be instantaneous (that is, the task switching time is zero). Each point presented in the figures below are average values obtained from 100 executions, with each execution running for 80000 task invocations. In order to ensure that the comparison is fair, the admission controllers for each of the four schedulability analysis techniques are allowed to run on the same 100 task sets. The 95% confidence interval of the values presented are within 1% of the mean, and are not shown in the figures for the sake of legibility.

First we compare the four schedulability tests for the admissible utilization for different number of nodes in the system, the results of which are shown in Figure 8. We consider
system sizes ranging from 5 nodes to 25 nodes. Both network calculus and holistic analysis perform poorly when the system size increases, and the drop in their admissible utilization is steeper than for the meta-schedulability test and the multi-modal analysis presented in this paper. We note that for small system sizes (up to 10 nodes), holistic analysis in fact, performs better than the multi-modal analysis. The reduction from the distributed system to the multi-modal uniprocessor assumes that each higher priority task invocation delays the lowest priority task at two stages (according to the delay composition theorem). However, not all higher priority task invocations interfere at two stages, and some cause a delay less than what is quantified by the delay composition theorem as the worst-case. For small system sizes, holistic analysis is able to determine the number of invocations of higher priority tasks that delay the lowest priority task under consideration more accurately. However, for large systems (more than 15 nodes), holistic analysis becomes extremely pessimistic and the multi-modal analysis performs better. The multi-modal analysis is able to admit about 25% more tasks than the next best analysis technique for large systems.

We next varied the probability with which a node is chosen to be part of a task’s route (the value $\text{node} \_\text{prob}$), and present the results in Figure 9. As the value of $\text{node} \_\text{prob}$ increases, task routes become longer, and both holistic analysis and network calculus become more pessimistic and their admissible utilization drops. The meta-schedulability test and its extension presented in this paper perform well for tasks with long routes, as the number of precedence constraints between successive stages of tasks that are relaxed become lower. For tasks with short routes, a larger fraction of the total number of constraints are relaxed leading to poorer performance. Thus, for both these tests, it is the fraction of precedence constraints that are relaxed that affects performance and not the length of task routes, unlike holistic analysis and network calculus. Further, when the $\text{node} \_\text{prob}$ value is small and tasks have short routes through the system, the problem with the meta-schedulability test explained in Section 4.3 becomes exacerbated. Higher priority periodic tasks delay a lower priority task only at a few stages. However, in the hypothetical uniprocessor system, the corresponding tasks are assumed to delay the lower priority task throughout its execution. This problem is overcome by the multi-modal uniprocessor model presented in this paper. In fact, for short task routes, the extension allows almost twice as many tasks to be admitted as compared to the meta-schedulability test. For strict pipelines (a $\text{node} \_\text{prob}$ value of 1), the analysis in this paper admits more than twice as many tasks as holistic analysis or network calculus admits. The test accurately estimates the parallelism in the execution of successive stages in the pipelined distributed system, and is able to perform significantly better than holistic analysis.

Next, we compare the schedulability tests for different deadline ratio parameter values. For small values of $\text{DR}$, the computation times of all the tasks are comparable. For larger values of $\text{DR}$ (closer to 3), the computation times of tasks are widely varying and lower priority tasks manage to execute in the background of higher priority tasks. This allows busy periods to be longer and the utilization of the system to be higher for all the schedulability tests. The analysis presented in this paper achieves an increase of 20-50%, compared to

![Figure 8: Comparison of average per stage utilization for different number of stages in the system](image)

![Figure 9: Comparison of average per stage utilization for different probabilities of node being part of a task’s route](image)

![Figure 10: Comparison of average per stage utilization for different deadline ratio parameter values](image)
the admissible utilization using holistic analysis.

![Figure 11: Comparison of average per stage utilization for different ratios of task periods to end-to-end deadlines](image)

All the experiments conducted so far assumed that the task periods are equal to their end-to-end deadline. We allowed the end-to-end deadlines to be progressively tighter and considered ratios of task periods to end-to-end deadlines to be 1.33, 1.5, 2.0, and 2.5, while keeping the task periods the same. As expected the admissible utilization of all the four analysis techniques dropped with increasing ratio values (decreasing end-to-end deadlines). Yet, the multi-modal analysis presented in this paper, significantly outperforms existing analysis techniques for all values of the ratio of task periods to end-to-end deadlines.

![Figure 12: Comparison of average per stage utilization for different task resolution parameter values](image)

Finally, we conducted experiments that varied the task resolution parameter values, that is, the ratio of the computation times of tasks to their end-to-end deadline. The average per stage utilization for the four admission control tests for task resolution parameter values of 1/20, 1/40, 1/60, 1/80, and 1/100 are shown in Figure 12 (note that the x-axis shows 1/task resolution). The task resolution parameter does not affect the performance of the various tests, showing that the tests are not sensitive to the size of the tasks. This is due to the preemptive nature of scheduling. A task resolution parameter of 1/100 would approximately have five times as many tasks admitted as a task resolution parameter of 1/20, but each task would be five times smaller in terms of computation time, overall resulting in approximately the same interference to the lowest priority task.

6. Related Work

Techniques such as [21, 5] to statically schedule distributed tasks by constructing a schedule of length equal to the least common multiple of the task periods have been proposed. Clearly, these techniques have exponential time complexity and are not suited for large systems. Pipelined distributed systems have been studied in the context of job-fair scheduling, to determine if its feasible to schedule a given set of tasks on a set of pipelined resources. For instance, [2] studies special cases of the problem, and provides heuristic solutions to the general case. In contrast, in this paper, we are interested in the schedulability problem of a given set of tasks in a distributed set of resources scheduled using a fixed priority policy.

Several offline schedulability tests have been proposed that divide the end-to-end deadline into per-stage deadlines such as [11, 22]. Each stage is then considered independently of the others, and if every stage satisfies the per-stage deadlines of all the tasks, then the system is deemed to be schedulable. However, such an approach largely underestimates the inherent parallelism in the execution of different pipeline stages and tends to be extremely pessimistic for systems with several resource stages.

The two main analysis techniques used for analyzing end-to-end delay in distributed systems are techniques based on holistic analysis and network calculus. Both these techniques are based on the basic principle that one can determine the exit pattern of jobs leaving a particular resource using the arrival pattern of jobs to that resource. The exit pattern of jobs from one resource, in turn becomes the arrival pattern of jobs to the next resource. By successively applying this process, the worst-case end-to-end delay of jobs can be computed. Holistic analysis was first proposed in [19], and has since been improved and generalized. For instance, [12, 13] propose offset-based response time analysis techniques, which considers the offsets of tasks in addition to their jitter. Network calculus was proposed in [3, 4], and provided a means to determine the end-to-end delay of packets of flows. This was applied to real-time systems, called Real-Time Calculus, first presented in [17]. Several works such as [9, 20], have extended and generalized this technique. Nevertheless, both holistic analysis and network calculus based approaches suffer from the fundamental problem of being increasingly pessimistic with system scale, and become complex or even impossible to use for large systems.

To address the problem of scalability, a reduction-based approaches were presented in [6, 7, 8]. Here, the schedulability problem on a distributed system was reduced to that on a hypothetical uniprocessor. This enabled any uniprocessor analysis technique to be used to analyze pipelined systems, and the solution was no longer increasingly pessimistic for large systems. However, this solution assumed
that each higher priority task interfered with a lower priority task throughout its execution on the uniprocessor. In contrast, in the original distributed system, each higher priority task delays a lower priority task only on stages where both tasks execute. The solution, thus, needlessly overestimates the worst-case delay. To overcome this problem, in this paper, we present a reduction of the distributed schedulability problem to a hypothetical multi-modal uniprocessor system. The transition of a task from one resource to another in the distributed system, encountering a different set of higher priority tasks at each resource, is modeled as mode changes in the uniprocessor. With this enhanced uniprocessor model, we provide much tighter analysis for periodic tasks that scales well with system size.

Unlike traditional multi-modal uniprocessor analysis [16, 18, 15, 14], the model presented in this paper is unique in that the time instants of the mode changes are not known apriori. The problem therefore becomes one of estimating the response time for the worst-case scenario of mode changes in the uniprocessor, a problem not handled previously.

7. Conclusion

In this paper, we define a new multi-modal uniprocessor system model that is inspired by distributed task set transformation. We show that the end-to-end delay analysis of a task in a distributed system, can be reduced to that of analyzing an equivalent hypothetical multi-modal uniprocessor. From the perspective of a distributed task, it encounters a different set of higher priority tasks at each resource stage. This is modeled as mode changes in the uniprocessor. Thus, when the distributed task under consideration moves from one resource to another in the original distributed system, a mode change is triggered in the hypothetical uniprocessor. Unlike classical literature on mode changes, our model has no prior information on the precise timing of the mode changes. Therefore, the objective is to estimate the response times of tasks in the uniprocessor subject to any worst-case scenario of mode changes. We present an iterative solution using dynamic programming that converges to an upper bound on the worst-case response time, as long as the system utilization during any mode remains within 100%. Simulation studies suggest that the analysis presented in this paper admits over 25% more per-resource utilization compared to existing analysis techniques for distributed systems.

References