RC24503 (W0803-004) March 3, 2008
Computer Science

IBM Research Report

Online Optimization for Latency Assignment in Distributed Real-Time Systems

Cristian Lumezanu
University of Maryland

Sumeer Bhola
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Mark Astley
Two Sigma Investments, LLC

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Online Optimization for Latency Assignment in Distributed Real-Time Systems

Cristian Lumezanu[*]
University of Maryland
lume@cs.umd.edu

Sumeer Bhola
IBM T.J. Watson Research Center
sbhola@us.ibm.com

Mark Astley[*]
Two Sigma Investments, LLC
mark.astley@twosigma.com

Abstract

As distributed real-time applications gain in popularity, a key challenge is to allocate resources so that diverse real-time requirements (including non-real-time applications), distributed application components and varying workloads can all be accommodated without violating timeliness constraints. We examine the problem of resource allocation in distributed soft real-time systems, where both network and CPU resources are consumed. The timeliness constraints of applications are expressed through utility functions, which compute "benefit" as a function of end-to-end latency. We present LLA *(Lagrangian Latency Assignment), a scalable and efficient distributed algorithm which maximizes aggregate utility by computing an optimal trade-off between end-to-end latency and allocated resources. The algorithm runs continuously and adapts to both workload and resource variations.* LLA *is guaranteed to converge if the workload and resource requirements stabilize. We evaluate the quality of results and convergence characteristics under various workloads, using both simulation and real-world experimentation.*

1. Introduction

Recent years have witnessed the emergence of distributed real-time systems as a platform for enterprise applications seeking to respond rapidly to real world events. Representative examples include program trading, risk management, medical alerting, patient monitoring, airline ticket pricing and environmental monitoring. The common theme across all these applications is that they leverage *distributed and heterogeneous resources* to *continuously* process and analyze real-world data in *real-time* and to build accurate models used for prediction or prevention. Consider, for example, a program trading application which trades securities based on real-time analysis of market data. Bandwidth and CPU are both constrained resources in such an application: available network bandwidth must be balanced between receiving market data, and issuing trades; and CPU must be balanced between handling market data and performing trading strategy analysis.

Ensuring that the real-time requirements of distributed applications are satisfied is challenging. First of all, due to technology convergence, applications with a diversified set of real-time demands share the same infrastructure. Thus, a scheduling algorithm should be *flexible* in accommodating different quality of service requirements and in quantifying the importance of applications relative to each other. Second, the available resources are not dedicated and may change over time due to failures. Workloads may also vary, typically because communication is triggered by real world events. Overprovisioning is not feasible since it has significant cost in terms of hardware, space, power and human resources. Instead, it is preferable to manage the scheduling of existing resources *dynamically*. For example, in program trading, a dynamic strategy which is work conserving is particularly favorable as certain functions, such as trading strategy analysis, can use surplus resources to their advantage. Managing resources statically, on the other hand, may lead to underutilization or starve important functions during high load.

In this paper we present LLA (Lagrangian Latency Assignment), a distributed feedback-based optimization algorithm to control the scheduling parameters for soft real-time applications in a distributed system, such that

[*] Work done while authors were at the IBM T.J. Watson Research Center.

the aggregate system utility is maximized. There has been much recent work in the real-time community on feedback control approaches for scheduling sets of distributed applications. However, this research (see Section 7 for a detailed comparison) is typically limited to adjusting the aggregate CPU utilization on servers to ensure that all the distributed applications are schedulable. It does not take into account flexible application deadlines, different levels of importance for applications or network bandwidth resources. LLA, on the other hand, *incorporates limits on both CPU and network bandwidth*, and in general can accommodate any similar resource constraints. Moreover, our approach *specifies the utility of the system as a non-increasing function of the latency of each application*, which implicitly expresses the application importance with respect to other applications as well as the importance of meeting a particular latency requirement. Furthermore, it allows the use of different percentiles of individual latencies when computing the utility function. For instance, one application may use a 99^{th} percentile of all its individual latencies, while another may use a 50^{th} percentile, depending on the nature of the application or its SLA.

The optimization problem is solved on-line, in a distributed manner, using the "price" of resources to coordinate the resource consumption by different applications. As the optimization is constantly running, the system is adaptive, and adjusts to both workload and resource variations. The algorithm is guaranteed to converge if the workload and resource requirements stabilize. Our optimization approach is model-based. For each application and resource used by that application, we predict the latency at that resource (the latency slice contributing to the end-to-end latency) as a continuous function of the scheduling parameter. The scheduling parameter is a proportion of the resource allocated (*i.e.*, we assume proportional share scheduling), but no particular implementation is mandated. The model itself can be constructed on-line, and iteratively improved as the system is running, as we show in our experiments.

The main contributions of this paper are:

1. A framework for unifying diverse real-time requirements, using utility functions as *metrics* to measure application importance, into an objective for the system to achieve

2. A distributed algorithm that continually optimizes the system utility, by adjusting scheduling parameters, and which takes into account feedback of workload, resource and model changes. Under certain constraints, the algorithm is guaranteed to converge to the optimal solution.

3. An experimental evaluation using both simulation and a prototype implementation, that demonstrates fast convergence, scalability, and the ability to improve the latency models at runtime.

The paper is structured as follows. We present a programming model for real-time applications in Section 2. Next, we define the optimization problem (Section 3) and describe the LLA algorithm (Section 4). We evaluate LLA through extensive simulations (Section 5) and real system experimentation (Section 6). We review related work is Section 7 and we conclude in Section 8.

2. Programming Model

We consider distributed real-time applications that can be modeled using the typical task and subtask model, with the generalization that multiple jobs in a subtask can be released without waiting for previous jobs to finish. This generalization captures real-life workloads with bursty arrivals. The context for our model is a distributed system composed of nodes interconnected by links. Each node and link provides a set of resources for which applications compete in order to meet timeliness constraints. For example, nodes provide CPU, whereas links provide network bandwidth.

Applications are defined similar to an end-to-end task model [31] in which there are a set of *tasks*, $\mathcal{T} = \{T_i\}$, each of which consists of a set of *subtasks*, $\mathcal{S}_i = \{T_{ij}\}$. Subtasks may utilize different resources. For simplicity of exposition and without loss of generality, we impose the restriction that each subtask consumes exactly one resource.

Subtasks may also specify properties which describe how a resource will be utilized, for example worst case execution time (WCET). Note that an application consisting of computation and communication will be modeled uniformly in terms of subtasks: computation is modeled as subtasks which consume processor resources; and communication is modeled as subtasks which consume network resources.

Tasks are dispatched/released in response to *triggering events* which are signals with an arrival pattern and optional data. For example, a triggering event may be a periodic signal at a constant rate. The arrival patterns of triggering events are included in task specifications, or measured at runtime, for scheduling purposes.

The release of subtasks is constrained by a precedence relation called a *subtask graph*, which is a directed acyclic graph of subtasks with a unique root. The root is called the *start subtask*, and the leaf nodes are called *end subtasks*. Edges in the graph represent precedence, either in the form of data transmission or logical ordering constraints. Formally, the subtask graph, $\mathcal{G}_i$, for task T_i is denoted by the relation $\mathcal{G}_i \subset \mathcal{S}_i \times \mathcal{S}_i$ where $\mathcal{G}_i$ is

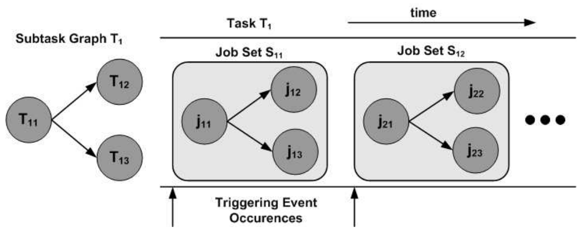

Figure 1. A task T_1 consists of a subtask graph and a set of triggering events. When a triggering event occurs, the first subtask in the subtask graph becomes eligible for release. The job sets and jobs resulting from two task releases are shown in the time-line on the right.

acyclic. A path p in $\mathcal{G}_i$ is defined to be a sequence of subtasks $T_{ia}, T_{ib}, \ldots, T_{in}$ where:

- each adjacent pair $(T_{ix}, T_{iy}) \in \mathcal{G}_i$;
- T_{ia} is the unique root of $\mathcal{G}_i$; and
- T_{in} is a leaf node of $\mathcal{G}_i$.

We denote by $\mathcal{P}_i$ all paths in the subtask graph of a task i.

We use the term *job* to distinguish separate instances of a released subtask. As mentioned earlier, jobs of a subtask can be released concurrently or overlap. Regardless of overlap, precedence constraints across subtasks must still be observed. The set of jobs which correspond to a particular task release are called a *job set* and represent an instance of the subtask graph. Formally, a job set $\mathcal{J}_{ij}$ represents the j^{th} instance of task T_i and consists of a set of jobs $\mathcal{J}_{ij} = \{j_{jk} : T_{ik} \in \mathcal{S}_i\}$. Figure 1 illustrates these concepts.

Task execution is subject to timeliness constraints which we describe below.

2.1. Timeliness Constraints

The timeliness constraint for a task limits the total latency incurred by a job set dispatched for the task. The latency for a job set is defined as the interval between the dispatch time of the root subtask and the completion time of all eligible end subtasks. We specify this timeliness constraint using a *utility function* which is a non-increasing function that maps job set latency to a utility value. The maximum allowable latency may be limited by a *critical time* beyond which latency may not extend regardless of utility. Thus, critical time is analogous to a deadline.

Utility functions are a generalization of simple deadlines where, in addition to defining absolute constraints (*e.g.*, latency must not exceed the critical time), the shape of the function can be used to derive trade-offs between latency (*i.e.*, resource allocation) and benefit to the application. Thus, our goal is to satisfy all application deadlines (*i.e.*, critical times) while maximizing utility.

The latency (and hence utility) of a job set depends on the latency experienced by the individual jobs within the set. The latency experienced by an individual job depends on resource allocation and may vary according to application parameters. Task specifications are expected to define properties which help to determine the latency for jobs (*e.g.*, worst case or average case execution time). Specifications could be derived or corrected from runtime measurements. We can combine these specifications (including trigger event specifications) together with a model of resources to derive the predicted latency for a job.

When job latency is worst case, we can formalize utility computation as follows (we consider other than worst case below). Let T_i be a task with subtasks $\mathcal{S}_i$ and subtask graph $\mathcal{G}_i$. For a subtask $s \in \mathcal{S}_i$, let $\texttt{lat}_s$ be the worst case latency for any release of s given current resource allocations.

The (worst case) latency of a path, $p \in \mathcal{P}_i$ is the sum of the latencies of each subtask in the path: $\sum_{s \in p} \texttt{lat}_s$. We define the *critical path* as the path with the maximum latency among all possible paths in a subtask graph. Thus, the (worst case) latency of a job set is the latency of the critical path. Therefore, the utility for a task T_i is given by the function:

$$U_i = f_i(\max_{p \in \mathcal{P}_i} \sum_{s \in p} \texttt{lat}_s) \tag{1}$$

where examples of f_i are the functions shown in Figure 2. That is, utility is computed from the worst possible latency experienced for the task.

The case where $\texttt{lat}$ is other than worst case is more complicated. Let $\texttt{lat}_s^p$ be the latency bound for the p^{th} percentile of jobs released for subtask s. For example, $\texttt{lat}_s^{50}$ gives the median latency. Note that for two subtasks a and b, each with the same number of released jobs, the sum $\texttt{lat}_a^p + \texttt{lat}_b^p$ yields the $\frac{p^2}{100}$ latency percentile. Thus, if all paths have the same length n, we must use the $p^{\frac{1}{n}} \times 100^{\frac{n-1}{n}}$ latency percentile for each subtask in order to compute utility as a function of the p^{th} latency percentile. If path lengths are not identical, then separate latency functions must be used depending on the path being computed. Our model can be used with any latency percentile, but to simplify the exposition we will omit the percentile subscript and assume that the percentiles have been appropriately chosen for each subtask latency function. Also, for simplicity of exposition, we assume that no two subtasks in the same task consume the same resource.

3. Optimization Problem

Our goal is to find the latencies for each subtask in the system such that we achieve optimal value for the sum of utilities across all tasks. We express this goal as a constrained optimization problem.

3.1. Optimization

Let $\mathcal{R}$ be the set of all resources. Every resource is characterized by a *share function* to map subtasks to resource shares and an *availability* value. The resource availability, $B_r \in [0, 1]$, represents the fraction of the resource available to our competing tasks. We define the share function later in this section.

Each subtask is part of exactly one task and will utilize exactly one resource. For simplicity, we abuse notation and denote all subtasks associated with either a particular task or resource by $\mathcal{S}_i$ where i represents the task or the resource, depending on the context. Similarly, all resources where a task i executes are denoted by $\mathcal{R}_i$. Furthermore, unless we explicitly need to distinguish among separate instances of the same subtask or task, we use interchangeably the terms *job* and *subtask*, respectively *job set* and *task*. For every task i, C_i is the *critical time* (*i.e.*, deadline) of the task. Every subtask s has a predicted *latency* ($\mathtt{lat}_s$). The latency is determined by the resource the subtask utilizes using both subtask properties (*e.g.*, WCET) and resource properties (*e.g.*, lag in scheduling, share assignment).

Our objective is to maximize the total utility of the system, defined as the sum of utilities across all tasks:

$$\max \sum_{i \in \mathcal{T}} U_i \qquad (2)$$

There are two different constraints:

Resource Constraint. Each subtask competing for a resource receives a share of the resource. To model the correspondence between a subtask, its latency and its share, we define, for each resource r, the function $\mathtt{share}_r : \mathcal{S}_r \times \mathbb{R}^+ \rightarrow [0, 1]$. The resource constraint states that the sum of resource shares allocated to each subtask must be lower than the fraction of available resource:

$$\sum_{s \in \mathcal{S}_r} \mathtt{share_r(s, lat_s)} \leq B_r, \forall r \in \mathcal{R} \qquad (3)$$

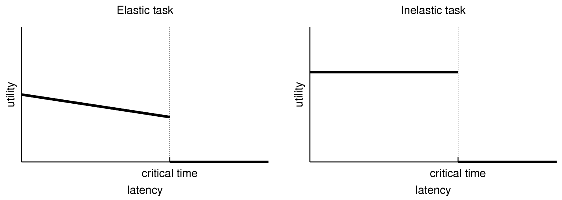

Figure 2. Utility functions for elastic and inelastic tasks.

Critical Time Constraint. To ensure that a task instance finishes in time, the end-to-end latency for each path in the subtask graph must be smaller than its critical time.

$$\sum_{s \in p} \mathtt{lat}_s \leq C_i, \forall i \in \mathcal{T}, p \in \mathcal{P}_i \qquad (4)$$

3.2. Discussion

Utility Functions. The utility of a task represents the benefit derived from completion of the task. Following the model of time-utility functions proposed by Jensen *et al.* [8], utilities are non-increasing functions that map the end-to-end task latencies to a benefit value. Figure 2 illustrates two examples of utility functions. The function on the left characterizes an *elastic* task in which benefit increases as latency decreases. Such tasks are typical of soft real-time systems and allow trade-offs between overall system benefit and utilization of resources. The function on the right characterizes an *inelastic* task and represents traditional hard real-time scheduling where the only important behavior is that tasks complete before their deadline. Inelastic tasks constrain resources, but do not allow trade-offs between benefit and utilization. Our approach can accommodate both elastic and inelastic tasks as long as utility functions are concave and continuously differentiable when latency is less than the critical time.

Equation 1 defines the task utility in terms of the critical path in the subtask graph. However, since the algorithm continuously adjusts the scheduling parameters to reflect the best possible allocation, the critical path may change. This behavior may make the objective function non-concave and may prevent the algorithm from finding a single optimal allocation. Therefore, to make the problem tractable, we propose two variations of the utility function of a task T_i:

- *sum*. The utility of T_i is a function of the sum of the latencies of each subtask belonging to T_i.
- *path-weighted*. The utility of T_i is a function of the weighted sum of the latencies of each subtask belonging to T_i. The weight of each subtask s is

proportional to the number of paths that s belongs to.

We believe that these variations are flexible enough to allow for a close approximation of the optimal allocation. As we show in Section 5, the critical path obtained when maximizing the path-weighted utility is always less than 1% smaller than the critical time (*i.e.*, the maximum possible critical path).

Proportional Share scheduling. We use a proportional share (PS) mechanism to map subtasks to share requirements. In proportional share scheduling, every subtask receives a fraction of the resource it utilizes. This fraction, the share, provides an easy way to partition CPU and link resources, and thus provides performance isolation between subtasks [25]. In the context of soft real-time systems, it is important to prevent poorly behaved subtasks from consuming more than their allotment of share. Furthermore, PS schedulers are work-conserving, so spare capacity can be utilized. Other mechanisms, like traditional priority or time-sharing schedulers, are simpler than proportional share, but do not offer a straightforward way for partitioning resources or enabling performance isolation.

Admission Control. Admission control could be used at the granularity of tasks, or for job sets in a task. We assume any admission control is layered on top of our approach, and is outside the scope of this paper.

4. Distributed Optimization Algorithm

In this section we describe LLA (Lagrangian Latency Assignment). LLA is a distributed algorithm, based on the Lagrange multiplier theory, that assigns latencies to all subtasks in the system such that the total system utility is optimal.

4.1. Overview

We assume there exists a task controller for each task in the system. Each controller determines the resource share and latencies for all subtasks that belong to the task. Task controllers may execute on dedicated nodes or may occupy the resources where the start subtasks of each task execute.

We say that the system is *congested* whenever at least one of the constraints defined by Equations 3 and 4 is violated. We identify two types of congestion, depending on the type of constraint that is not respected. Resource congestion occurs when a resource cannot schedule all subtasks executing locally (*i.e.* the sum of their shares is greater than B_r) and path congestion occurs when a path in the subtask graph cannot finish execution before its critical time.

At any moment, we can make the utility of a task higher by decreasing the latency of any of the subtasks on the critical path of the task. This may potentially create congestion in the system, both in a direct and an indirect way. First, decreasing the latency of a subtask makes the share allocated to the subtask bigger, which can lead to resource congestion. The only way to control the resource congestion is to give a smaller share to at least one of the other subtasks executed on the resource. However, decreasing the share of a subtask makes the individual latency of the subtask bigger and, if the subtask is on a critical path, can delay the associated task beyond its critical time. Thus, decreasing the latency can also create path congestion in the system. To guarantee that no congestion occurs, a task controller would have to be coordinated with all the other task controllers, which is impractical in real systems. We use the concept of price [13, 18] to solve the problem in a distributed setting. A price is associated with each resource and each path and indicates the level of congestion in the resource or the path. Each resource[1] computes a price value and sends it to the controllers of the tasks that have subtasks executing at the resource. Each controller computes prices for all paths in the associated tasks. Based on the received resource prices and the local path prices, a controller can calculate new latencies for the subtasks in its task.

LLA solves the optimization problem iteratively. A single iteration consists of **latency allocation** and **price computation**. Latency allocation predicts the optimal latencies at a certain time, given fixed resource and path prices. Price computation computes new values for the prices, given constant latencies for all subtasks in the system. The algorithm iterates indefinitely but the allocations may be only enacted periodically or when significant changes occur.

4.2. Latency Allocation

The latency allocation algorithm runs at each task controller and computes new latencies for all subtasks in the task, based on feedback from the resources where these subtasks run and from the paths to which they belong. Latencies are computed using the Lagrangian [4] of the original optimization problem (Equations 2,3 and 4):

[1] Prices for link resources are computed by one of the endpoints of the link.

$$
\begin{aligned}
L(\texttt{lat}_s, \mu_r, \lambda_p) &= \sum_{i \in \mathcal{T}} U_i \\
&- \sum_{r \in \mathcal{R}} \mu_r \big(\sum_{s \in \mathcal{S}_r} \texttt{share}_\texttt{r}(\texttt{s}, \texttt{lat}_\texttt{s}) - B_r \big) \\
&- \sum_{i \in \mathcal{T}, p \in \mathcal{P}_i} \lambda_p \big(\sum_{s \in p} \texttt{lat}_s - C_i \big) \qquad (5)
\end{aligned}
$$

where μ_r and λ_p are the Lagrange multipliers and can be interpreted as the price per unit of resource r and path p, respectively. We will simply refer to μ_r as resource price and to λ_p as path price.

We assume that the utility functions, expressed in terms of subtask latencies, are concave and continuously differentiable, in the region where the critical time constraint is satisfied (Equation 4). We also assume that the share functions are strictly convex and continuously differentiable, since increasing latency leads to diminishing returns in terms of decreasing share (and vice versa). This implies that if the utility functions were expressed in terms of share allocation, they would be strictly concave and continuously differentiable. This strict concavity, along with the fact that the resource constraints and critical time constraints are convex sets, means that finding the maximum for the objective function is equivalent to finding the maximum for the Lagrangian (dual problem) [4]. Thus, instead of solving the original optimization problem, we solve the following alternative problem for each task i, given specific values for μ_r and λ_p:

$$
\begin{aligned}
D(\mu_r, \lambda_p) = \max_{\mathcal{L}_i} L(\texttt{lat}_s, \mu_r, \lambda_p) \\
\mathcal{L}_i = \{\texttt{lat}_s, \forall s \in \mathcal{S}_i\}, r \in \mathcal{R}_i, p \in \mathcal{P}_i \qquad (6)
\end{aligned}
$$

Based on the earlier assumptions, the objective function in Equation 6 is strictly concave and continuously differentiable. The maximum is found by setting its derivative with respect to each $\texttt{lat}_s \in \mathcal{L}_i$ to 0:

$$
\frac{\partial L}{\partial \texttt{lat}_s} = \frac{\partial U_i}{\partial \texttt{lat}_s} - \sum_{p \in \mathcal{P}_i} \lambda_p - \mu_r \frac{\partial \texttt{share}_\texttt{r}(\texttt{s}, \texttt{lat}_\texttt{s})}{\partial \texttt{lat}_s} \qquad (7)
$$

where i is the task containing subtask s and r is the resource utilized by subtask s.

The latency allocation step is performed by executing the following algorithm at the controller of each task:

Latency Allocation

```
OUTPUT: Latencies $\texttt{lat}_s$, $\forall i \in \mathcal{T}, \forall s \in \mathcal{S}_i$, at iteration
        $t = 1, 2, ...$
1: Receive the resource price values $\mu_r$, $\forall r \in \mathcal{R}_i$.
2: Compute the path price values $\lambda_p$, $\forall p \in \mathcal{P}_i$.
3: Compute new latencies $\texttt{lat}_s$ by setting the derivative w.r.t. $\texttt{lat}_s$ of the Lagrangian to 0.
4: Send $\texttt{lat}_s$ to the resource where the corresponding subtask $s$ is executed.
```

4.3. Price Computation

Prices reflect the congestion of resources and paths. At every iteration, the newly computed latencies may potentially affect the schedulability of subtasks on resources or the end-to-end latencies of paths and thus may change the levels of congestion. Consequently, the resource and path prices need to be readjusted. The price computation consists of determining new values for the resource and path prices, given the latencies computed in the previous step. Resource prices are computed by each resource locally, while path prices are computed by the controller of the task to which the path belongs. We use price adjustment algorithms similar to those described by Low *et al.* [18]. They are based on the gradient projection method: prices are adjusted in a direction opposite to the gradient of the objective function of the dual problem (Equation 6). The component of the gradient corresponding to the prices μ_r, $\frac{\partial D}{\partial \mu_r}$, represents the available fraction of resource r. Similarly, $\frac{\partial D}{\partial \lambda_p}$ is the available time the end-to-end latency of path p can afford to increase (*i.e.*, slack of the path).

The resulting formulas for adjusting resource and link prices are:

$$
\mu_r(t+1) = \mu_r(t) - \gamma_r (B_r - \sum_{s \in \mathcal{S}_r} \texttt{share}_\texttt{r}(\texttt{s}, \texttt{lat}_\texttt{s})) \qquad (8)
$$

$$
\lambda_p(t+1) = \lambda_p(t) - \gamma_p (1 - \frac{\sum_{s \in \mathcal{S}_p} \texttt{lat}_s}{C_i}) \qquad (9)
$$

where $r \in \mathcal{R}$, $i \in \mathcal{T}, p \in \mathcal{P}_i$; and γ_r, γ_p are step sizes, with $\gamma_r, \gamma_p \in [0, \infty)$. The step sizes control how aggressive the price updates are and implicitly, how much latencies vary. Intuitively, large step sizes trigger more aggressive updates and speed up the convergences, but may lead to oscillations. On the other hand, smaller step sizes slow the algorithm significantly, but produce less oscillations. In Section 5 we show how to adaptively choose step sizes based on resource congestion.

The resource price computation algorithm is shown below (the path price computation is similar):

4.4. Discussion

For the experiments conducted in this paper, the latency for subtasks is the worst-case latency. The share function is modeled on the worst case execution time of

Resource Price Computation

```
OUTPUT: Resource price $\mu_r$, for resource $r$, at iteration $t = 1, 2, ...$
  Receive the computed latencies of all subtasks running at $r$
  Compute a new resource price $\mu_r$ based on Equation 8
  Send the price $\mu_r$ to the controllers of tasks that have subtasks running at $r$
```

subtasks (c_s), latency of the subtask ($\texttt{lat}_s$), and the resource lag (l_r) due to PS scheduling. The share can be computed as:

$$\texttt{share}_\texttt{r}(\texttt{s}, \texttt{lat}_\texttt{s}) = \frac{c_s + l_r}{\texttt{lat}_s} \tag{10}$$

Conversely, we can predict the latency of a subtask if we know its share of the resource. Since the worst case execution time and the lag are fixed, the share varies only with the latency. Subtasks with smaller shares take longer to execute, while subtasks with bigger shares will have smaller latencies.

We have described LLA as a distributed iterative algorithm. It is interesting to consider how often we should enact the allocations. LLA runs continuously and adapts as the models or other aspects of the system change. However, new allocations are computed and enacted only when significant changes occur. For example, in our prototype experiments, described in detail in Section 6, the optimization algorithm executes much less frequently than regular processing, yielding low computation overhead. To minimize network overhead from the regular exchanges of prices between task controllers and resources, we can run the algorithm in batch mode, stopping it after it converges.

5. Simulation Experiments

In this section, we evaluate our optimization algorithm through simulation. We observe convergence properties by measuring algorithm performance over several workloads and using different formulations of the utility function.

5.1. Workload

We have constructed several test workloads by specifying a set of tasks and their characteristics.

The basic test workload has three tasks, as shown in Figure 4. Each of the three tasks is intended to mirror one type of distributed application with real-time requirements. The first task follows a push-based model similar to the publish/subscribe and multicast paradigms. In such a model, a distributed computation consists of a few nodes producing information and propagating it to all interested nodes. The second task represents a complex pull-based model employed by applications such as sensor-based systems or RSS feeds. The distributed computation starts with a node requesting information, aggregating it and sending it to other nodes. Finally, the third task is meant to represent a simpler pull-based model used in client-server applications. All three tasks are triggered by periodic events occurring every 100ms. Their end-to-end deadlines (critical times) are respectively 45, 76 and 53ms. Every task consists of several subtasks, each utilizing a different resource—either CPU or network bandwidth. The parametrization of the subtasks is given in Table 1. Ignore the rows corresponding to latency and critical path for now. We chose the parameters such that all resources are close to *congestion*: for every resource r, the sum of the shares received by each subtask running on r is close to B_r. The performance of LLA when resources are close to congestion constitutes a lower bound for its performance with all other schedulable workloads. We experiment with both utility variations discussed in Section 3.2: *sum* and *path-weighted*.

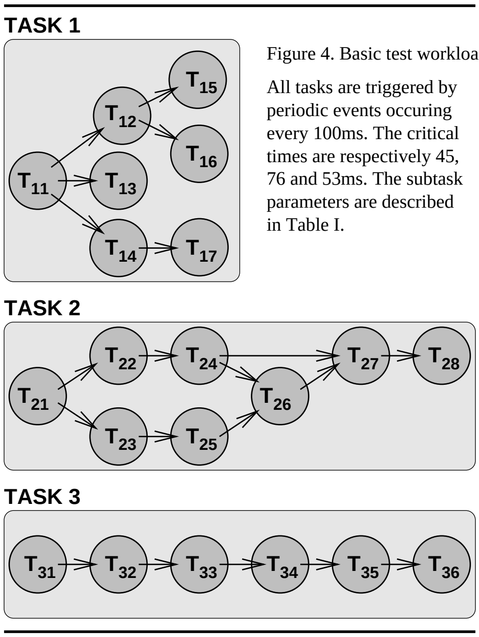

Figure 4. Basic test workload.

All tasks are triggered by periodic events occuring every 100ms. The critical times are respectively 45, 76 and 53ms. The subtask parameters are described in Table I.

5.2. Convergence

First we focus on the convergence properties of the algorithm. We use the *path-weighted* variation for the utility function:

$$U_i = f_i(\sum_{s \in \mathcal{S}_i} w_s \times \texttt{lat}_s) \tag{11}$$

	TASK 1							TASK 2								TASK 3					
	T_{11}	T_{12}	T_{13}	T_{14}	T_{15}	T_{16}	T_{17}	T_{21}	T_{22}	T_{23}	T_{24}	T_{25}	T_{26}	T_{27}	T_{28}	T_{31}	T_{32}	T_{33}	T_{34}	T_{35}	T_{36}
Resource	0	1	2	3	4	5	6	0	1	2	4	5	6	3	7	0	1	2	4	6	7
Exec time	2	3	4	5	4	3	2	2	4	3	6	7	5	2	3	3	2	2	3	4	4
Latency	9.7	13.8	19.5	14.4	21.4	10.5	19.2	10.3	15.0	15.1	19.3	12.8	16.6	5.1	9.3	9.9	7.9	6.2	9.8	10.3	8.7
Crit.Time	45							76								53					
Crit.Path	44.9							75.6								52.8					

Table 1. Task Parameters and Optimization Results (Execution time, latency, critical time and critical path are measured in milliseconds)

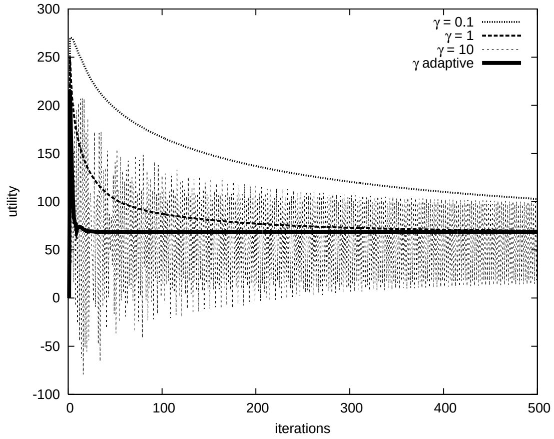

Figure 5. The effect of fixed and adaptive step sizes

The weight w_s of a subtask is equal to the number of paths in the task that the subtask belongs to. To map latency to benefit we use a simple linear continuous function: $f_i(\texttt{lat}) = k * C_i - \texttt{lat}$, where $k \geq 1$. In the experiments, we chose $k = 2$. Other values of k and other shapes of the utility yield similar results. We run the simulation four times, each time stopping it after 500 iterations. An iteration consists of a latency allocation run by each task controller and a resource allocation run at each resource. We measure the global value of the utility after each step. Since we want a fair trade-off between resource allocation and latency, we assume that the resource and path step sizes (γ_r and γ_p) are equal to each other and denote them simply by γ. At first, we assign fixed values to the step size. Later, we show how to adaptively change the value of γ. We experimented with several different values of the step size. Figure 5 depicts the system utility for three of them: 0.1, 1, and 10 (ignore the line corresponding to adaptive γ for now).

When the step size is high ($\gamma = 10$), the value of the utility oscillates with high amplitude around 50. If we decrease the step size ($\gamma < 10$), the utility converges. The number of iterations needed to achieve convergence depends on the value of γ. When $\gamma = 0.1$, the stabilization occurs after more than 1000 iterations (not shown in the figure), while for $\gamma = 1$, convergence is achieved after around 500 iterations. Thus, larger values of the step size lead to faster convergence, but they also make the oscillations larger. To turn this trade-off to our advantage, we should start with large step size values to ensure fast convergence. Then, we should decrease γ to minimize the size of the fluctuations.

We have implemented the following heuristic, based on experimentation, to adaptively change the value of the step sizes for resources and paths:

1. start with a fixed value for γ

2. at each iteration, if resource r is congested, double the step size associated with r, as well as the step sizes of all paths that traverse r

3. as soon as r becomes uncongested, revert the step sizes to the initial values

As long as a resource is congested, we increase multiplicatively the step sizes associated with it to speed up the convergence of the algorithm. When the resource becomes uncongested, we need more fine-grained updates to determine the convergence point, therefore we make the step sizes small again. We experimented with different starting values for the step size and we obtained the best results for $\gamma = 1$. We compare these results with those for fixed step size in Figure 5. The utility stabilizes faster and to a better value when the heuristic for adaptive γ was used. In Table 1 we show the subtask latencies and the task end-to-end latencies corresponding to the optimal utility. Each task completes execution before its critical time is reached.

We also tested the algorithm using the *sum* variation for the utility function but the results were not different in terms of convergence properties. From now on, unless specified otherwise, all results will be presented for an adaptive γ with *path-weighted* variation for the utility and $f_i(\texttt{lat}) = 2 * C_i - \texttt{lat}$ as utility function.

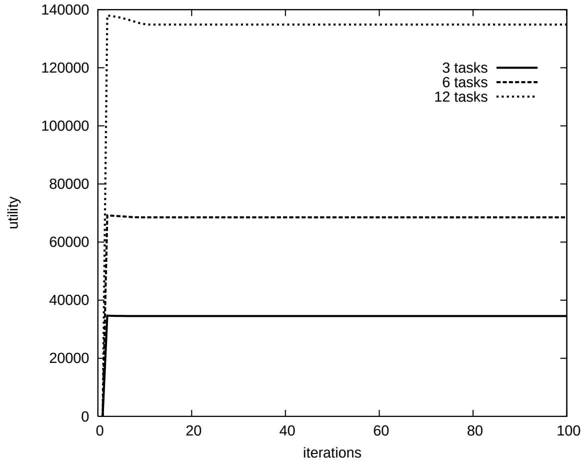

Figure 6. The effect on convergence as we scale the number of tasks

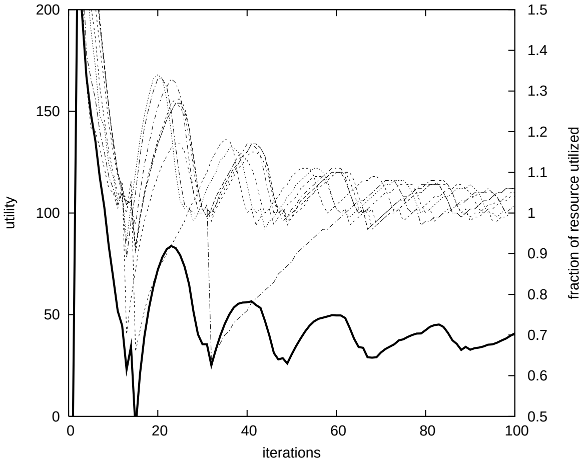

Figure 7. Using LLA to test the schedulability of a workload

5.3. Scalability

We verify if the algorithm maintains its convergence properties as we scale the number of tasks that execute simultaneously in the system. We start with the base workload and for each of the tasks we add another task with the same characteristics (subtasks, subtask parameters, subtask graph, subtask-to-resource mapping). Thus we obtain a workload with 6 tasks. We repeat the experiment to increase the number of tasks to 12 and we run LLA for the three workloads. However, as we add more tasks, more subtasks will contend for the same resources and the workload may become unschedulable. We ensure that schedulability is maintained by overprovisioning the system (*e.g.*, we set a high enough critical time for each task in all three workloads). Since the utility depends on the critical times, this will also produce higher values for the utility of each task. The results, presented in Figure 6, show that the convergence speed of the algorithm does not depend on the number of tasks executing simultaneously and that the value of the utility increases linearly with the number of tasks.

5.4. Workload Schedulability

LLA can be used to test the schedulability of a workload with respect to given system resources. An unschedulable workload should not show convergence of utilities, or the resource and critical time constraints should not be satisfied. As an experiment, we used the scaled six task workload from the previous section, but did not scale the critical times (the critical times were the same as for the three task workload). Figure 7 shows the total system utility (solid line) and the sum of shares for each resource as the number of iterations increase. Even after 100 iterations the utility and shares have not converged (compare with the convergence in figure 6). However, the fluctuations are slowly dampening, so one could conclude that this shows slow convergence. This conclusion is invalidated by observing the critical path latencies for the tasks and comparing with the critical time constraints. Across all the tasks, the critical path latencies are between $1.75 - 2.41$ times the constraint. For instance, for task 1, the critical path latency is 79ms, while the constraint is 45ms. So the system is not converging to a solution.

6. System Implementation

In this section, we describe a prototype implementation of our approach and its evaluation under a sample workload. In particular, we describe a mechanism for accommodating differences between predicted (*i.e.*, modeled) and actual performance.

6.1. Prototype

In order to test our approach under more realistic conditions, we have implemented a Java prototype which uses LLA to assign resources for Java-based tasks. The prototype executes on a virtual machine which supports the Real-Time Specification for Java (RTSJ) and includes IBM's Metronome Real-Time Garbage Collector [3]. The virtual machine executes atop IBM's Linux with Real-Time support (IBM-RTLinux), which has been further modified to allow share scheduling of the CPU. IBM-RTLinux is based on RedHat Enterprise Linux 4 and offers additional capabilities such as priority inheritance. The share scheduling support in the kernel implements a modified version of Surplus Fair-Share Scheduling [6].

6.2. Workload

The workload consists of four tasks, each with three subtasks, which are linearly dependent [2]. Each of the subtasks in a task runs on a different CPU resource, and there are a total of three CPU resources in the system. Therefore each CPU resource has 4 subtasks competing for it, one from each task. The subtasks are characterized using WCET, and a periodic arrival rate. Tasks 1, 2 have identical characteristics, and similarly tasks 3, 4 are identical. All subtasks of tasks 1, 2 have WCET of 5ms, and arrival rate of 40/second, and tasks 3, 4 have subtasks with WCET of 13ms, and arrival rate of 10/second. All tasks have the same utility function, $f_i(\texttt{lat}) = -\texttt{lat}$, with tasks 1, 2 having a critical time of 105ms, and tasks 3, 4 a critical time of 800ms. For the remainder of this discussion, we will refer to tasks 1, 2 as the *fast* tasks and tasks 3, 4 as the *slow* tasks.

Based on the arrival rate and WCET, the minimum share needed by each subtask of the fast tasks is 0.2 ($\frac{40}{second} \times 5ms$) and that by each slow subtask is 0.13. This is the share needed to keep up with the workload (otherwise jobs will queue up in an unbounded manner). So the sum of the minimum shares at each CPU is $0.2*2+0.13*2 = 0.66$ (66%), and this is also the CPU utilization due to this workload. In addition, a share of 0.1 was given to the Metronome garbage collector.

The network was not a constraint in this experiment.

6.3. Online Model Error Correction

The share function for each subtask is the one described earlier in equation 10, with a resource lag of 5ms. This share function is not always accurate. One important source of inaccuracy is that the release time of jobs for different subtasks sharing the same resource may not be synchronized, which leads to over-prediction of latency. To overcome this we use a simple additive error correction model, and do exponential smoothing of the error value. The samples for error correction were collected periodically, and high percentile samples (greater than 90th percentile) were used.

6.4. Experimental Results

The goal is to demonstrate: (1) the optimization algorithm running in a real system, (2) the effectiveness of model error correction, and (3) the ability of the optimizer to change its allocation based on model error correction.

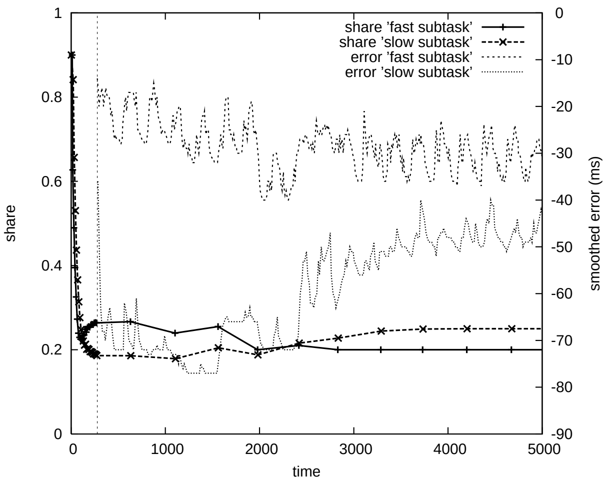

Figure 8. System experiment with model error correction

The optimizer runs continuously until the utility improvement from the previous iteration is below 1%. When the utility stabilizes, we run the optimization once a minute. To improve latency estimates, we perform error correction after every iteration. The computation overhead induced by the optimizer is rather small (below 1% of the total computation); the only observable overhead appears at the start of the optimization when there are large improvements in utility between iterations.

We start the experiment without model error correction, and let the optimizer compute share allocations based purely on the theoretical model. To meet the critical time constraint of the fast tasks the optimizer has to give them a higher share of the CPUs, and the remainder is allocated to the slow tasks. All the fast subtasks get the same share and similarly for the slow subtasks. A representative subtask is shown for each in figure 8. At time 277, shown by the vertical line in the figure, we enable error correction. At this instant the share of fast subtasks is 0.26 and slow subtasks is 0.19. Subsequently, the optimizer realizes that it can meet the critical time constraint of the fast tasks with a lower share, and eventually adjusts the allocation down to the minimum share for these subtasks (0.2 share). The remaining share is allocated to the slow subtasks (0.25 share). The percentage change in share allocation, due to error correction, is -23% and $+32\%$ for fast and slow subtasks respectively. The marginal resource cost of increasing the utility of the slow tasks is lower than that of the fast tasks, hence the former get a higher share. Note

[2] We use a smaller workload than the simulation experiments due to the lack of machine availability.

that the error value continues to fluctuate, however it exhibits some stability in its mean after the shares converge. This shows that additive error correction is not a perfect model (though possibly adequate), but despite that, the optimization algorithm converges to the optimal.

7 Related Work

Research related to our distributed optimization algorithm can be categorized as: deadline slicing in real-time systems, schedulability analysis and feedback-control scheduling, and network flow optimization.

Deadline slicing Deadline slicing techniques [21, 10, 9, 7, 26, 5, 11] try to find the deadlines that optimize a predefined measure of schedulability. These algorithms work with a fixed end-to-end deadline, limited characterizations of tasks (*e.g.*, periodic tasks, with WCET), and are offline.

BST [21] is a greedy breadth-first search algorithm that assigns slices—static execution windows in time—to tasks. The algorithm iteratively computes paths in the task graph that minimize the overall laxity, and assigns slices to the tasks by evenly distributing the path laxity. AST [10, 9] uses a task assignment algorithm which extends BST for the case where the resource to task mapping is not fixed *a priori*. Neither BST nor AST account for resource capacity. Garcia and Harbour [7] provide an iterative offline algorithm which assigns deadlines to sequential tasks. At each iteration, new deadlines are computed based on how far from schedulability each subtask is with its current deadline. Saksena and Hong [26] derive subtask deadlines by maximizing the critical scaling factor [16] of the task set. However, this algorithm works only with sequential tasks and as the authors assert, their solution is not optimal. Bettati and Liu [5] focus on distributed systems that can be characterized by a *flow shop* model: sequential subtasks execute on different resources in turn, following the same order. The authors assume identical execution times for all subtasks executing on the same resource and assign local deadlines by evenly distributing the end-to-end task deadline. In the context of soft real-time systems, Kao and Garcia-Molina [11] divide the deadline assignment problem into serial and parallel subtask problems. They propose simple strategies for both problems based on minimizing the deadline miss ratio.

The latency of a task in our framework can be interpreted as a soft deadline, that is then sliced to give individual soft deadlines for each subtask. Our algorithm produces an **optimal** latency assignment through **on-line** optimization. The objective function reflects different task importance and latency requirements.

Schedulability Analysis Meeting deadline requirements has traditionally relied on proper scheduling and schedulability analysis techniques. However, most scheduling algorithms focus on controlling performance on a single processor. Approaches such as Rate Monotonic and its extensions [16] use static allocation and assume *a priori* knowledge of the task parameters. In contrast, dynamic scheduling algorithms work with incomplete knowledge about the task set. These algorithms guarantee schedulability either by relying on information about the system resources [17], or through admission control and planning [23, 32]. Feedback-control scheduling [28, 19, 2, 24, 29] circumvents the problems of static and dynamic scheduling in unpredictable environments by continuous monitoring and adjustment of deadlines based on system feedback. Lu *et al.* [19] propose a feedback-control scheduling framework to minimize the deadline miss ratio of soft, independent tasks. Abdelzaher *et al.* [2] use utilization-based schedulability [1] to guarantee deadlines of aperiodic requests to a web server. Vengerov [29] proposes an approach based on reinforcement learning to schedule parallel jobs onto multiprocessor servers such that the average utility of completed jobs is increased. In the context of distributed systems, Stankovic *et al.* [27] use feedback-control scheduling to guarantee deadlines for sets of independent tasks that run on different processors.

Our work is most closely related to that of Lu *et al.* [20,31], in which utilization-based schedulability [1] is applied to schedule end-to-end tasks on a distributed platform. Lu *et al.* propose both centralized and distributed algorithms that control the invocation rate of tasks in order to adjust utilization of resources. Rate control can be considered a form of admission control, and is complementary to our approach of controlling latency (as opposed to rate). However, we consider much broader task behavior by allowing flexibility in latency sampling (*i.e.* percentiles), and task elasticity and importance (based on utility).

Network Optimization Several approaches for optimizing a system-wide utility function have been proposed in the area of network flow optimization. For practical reasons, distributed algorithms are highly desirable, often based on the dual decomposition [4, 22]. Relevant research includes work in unicast [18, 13, 14, 15] and multicast flow optimization [12, 30], in which flow rates are varied in order to optimize system utility. Our work is also based on dual decomposition, but defines system utility as a function of task latency (relevant for soft real-time applications) rather than flow rate. This formulation results in a somewhat different optimization problem, namely, we have nonlinear resource constraints due to the use of share scheduling and mapping latencies to shares.

8. Conclusions

In this paper we have framed the problem of latency assignment for soft real-time distributed applications as a utility optimization problem. Our framework allows us to accommodate flexible timeliness requirements, using various utility function shapes, and different percentiles of end-to-end latency. Moreover, this allows the system to make trade-offs based on different importance of applications. We have presented a novel distributed optimization algorithm that continuously optimizes the system, using feedback of resource congestion and latency constraints, and demonstrated fast convergence and scalability using both simulations and real system experiments.

References

[1] T. F. Abdelzaher and C. Lu. Schedulability analysis and utilization bounds for highly scalable real-time services. In *RTAS*, 2001.

[2] T. F. Abdelzaher, K. G. Shin, and N. Bhatti. Performance guarantees for web server end-systems: A control-theoretical approach. *IEEE Transactions on Parallel and Distributed Systems*, 13(1):80–96, 2002.

[3] D. F. Bacon, P. Cheng, D. Grove, and M. T. Vechev. Syncopation: generational real-time garbage collection in the metronome. In *LCTES '05*, pages 183–192, New York, NY, USA, 2005. ACM Press.

[4] D. Bertsekas. *Nonlinear Programming*. Athena Scientific, 1999.

[5] R. Bettati and J. W.-S. Liu. End-to-end scheduling to meet deadlines in distributed systems. In *ICDCS*, 1992.

[6] A. Chandra, M. Adler, P. Goyal, and P. Shenoy. Surplus fair scheduling: A Proportional-Share CPU scheduling algorithm for symmetric multiprocessors. In *OSDI 2000*, pages 45–58, 2000.

[7] J. J. G. García and M. G. Harbour. Optimized priority assignment for tasks and messages in distributed hard real-time systems. In *WPDRTS*, 1995.

[8] E. D. Jensen, C. D. Locke, and H. Tokuda. A time-driven scheduling model for real-time systems. In *Real-Time Systems Conference*, 1985.

[9] J. Jonsson. A robust adaptive metric for deadline assignment in heterogeneous distributed real-time systems. In *IPPS / SPDP*, 1999.

[10] J. Jonsson and K. G. Shin. Deadline assignment in distributed hard real-time systems with relaxed locality constraints. In *ICDCS*, 1997.

[11] B. Kao and H. Garcia-Molina. Deadline assignment in a distributed soft real-time system. *IEEE Transcations on Parallel and Distributed Systems*, 8(12):1268–1274, 1997.

[12] K. Kar, S. Sarkar, and L. Tassiulas. Optimization based rate control for multirate multicast sesions. In *IEEE Infocom*, 2001.

[13] F. P. Kelly, A. Maulloo, and D. Tan. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of Operations Research Society*, 49(3):237–252, 1998.

[14] S. Kunniyur and R. Srikant. End-to-end congestion control schemes: Utility functions, random losses and ECN marks. *IEEE/ACM ToN*, 11(5):689–702, 2003.

[15] R. J. La and V. Anantharam. Charge-sensitive TCP and rate control in the Internet. In *IEEE Infocom*, 2000.

[16] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *RTSS*, 1989.

[17] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[18] S. H. Low and D. E. Lapsley. Optimization flow control I: Basic algorithm and convergence. *IEEE/ACM ToN*, 7(6):861–874, 1999.

[19] C. Lu, J. A. Stankovic, S. H. Son, and G. Tao. Feedback control real-time scheduling: Framework, modeling, and algorithms. *Real-Time Systems Journal*, 23(1-2):85–126, 2002.

[20] C. Lu, X. Wang, and X. Koutsoukos. End-to-end utilization control in distributed real-time systems. In *ICDCS*, 2004.

[21] M. D. Natale and J. A. Stankovic. Dynamic end-to-end guarantees in distributed real time systems. In *RTSS*, 1994.

[22] D. P. Palomar and M. Chiang. On alternative decompositions and distributed algorithms for network utility problems. In *IEEE Globecom*, 2005.

[23] K. Ramamritham and J. A. Stankovic. Dynamic task scheduling in distributed hard real-time systems. *IEEE Software*, 1(3), 1984.

[24] B. Ravindran, P. Kachroo, and T. Hegazy. Adaptive resource management in asynchronous real-time distributed systems using feedback control functions. In *ISADS*, 2001.

[25] J. Regehr. Some guidelines for proportional share CPU scheduling in general-purpose operating systems. In *RTSS*, 2001.

[26] M. Saksena and S. Hong. An engineering approach to decomposing end-to-end delays on a distributed real-time system. In *WPDRTS*, 1996.

[27] J. A. Stankovic, T. He, T. Abdelzaher, M. Marley, G. Tao, S. Son, and C. Lu. Feedback control scheduling in distributed real-time systems. In *RTSS*, 2001.

[28] J. A. Stankovic, C. Lu, and S. H. Son. The case for feedback control real-time scheduling. Technical report, 1998.

[29] D. Vengerov. A reinforcement learning framework for utility-based scheduling in resource-constrained systems. Sun Microsystems Laboratory Technical Report 141, 205.

[30] W.-H. Wang, M. Palaniswami, and S. H. Low. Necessary and sufficient conditions for optimal flow control in multirate multicast networks. *IEE Proceedings-Communications*, 2003.

[31] X. Wang, D. Jia, C. Lu, and X. Koutsoukos. Decentralized utilization control in distributed real-time systems. In *RTSS*, 2005.

[32] W. Zhao, K. Ramamritham, and J. A. Stankovic. Preemptive scheduling under time and resource constraints. *IEEE Transactions on Computers*, 36(8), 1987.