Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research | Computer Science and Engineering

Report Number: WUCSE-2005-57

2005-11-01

End-to-End Scheduling Strategies for Aperiodic Tasks in Middleware

Yuanfang Zhang, Chenyang Lu, Christopher Gill, Patrick Lardieri, and Gautum Thaker

Many mission-critical distributed real-time applicationsmust handle aperiodic tasks with hard end-to-end dead-lines. Existing middleware such as RT-CORBA lacksschedulability analysis and run-time scheduling mecha-nisms that can provide real-time guarantees to aperiodictasks. This paper makes the following contributions to thestate of the art for end-to-end aperiodic scheduling in mid-dleware. First, we compare two approaches to aperiodicscheduling, the deferrable server and the aperiodic utiliza-tion bound, using representative workloads. Numerical re-sults show that the deferrable server analysis is less pes-simistic than the aperiodic utilization bounds when appliedoffline. Second, we propose a practical approach to tuningdeferrable servers for end-to-end tasks. Third, we describedeferrable server... **Read complete abstract on page 2.**

Follow this and additional works at: https://openscholarship.wustl.edu/cse_research

Part of the Computer Engineering Commons, and the Computer Sciences Commons

Recommended Citation
Zhang, Yuanfang; Lu, Chenyang; Gill, Christopher; Lardieri, Patrick; and Thaker, Gautum, "End-to-End Scheduling Strategies for Aperiodic Tasks in Middleware" Report Number: WUCSE-2005-57 (2005). *All Computer Science and Engineering Research.*
https://openscholarship.wustl.edu/cse_research/973

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

This technical report is available at Washington University Open Scholarship: https://openscholarship.wustl.edu/cse_research/973

End-to-End Scheduling Strategies for Aperiodic Tasks in Middleware

Yuanfang Zhang, Chenyang Lu, Christopher Gill, Patrick Lardieri, and Gautum Thaker

Complete Abstract:

Many mission-critical distributed real-time applicationsmust handle aperiodic tasks with hard end-to-end dead-lines. Existing middleware such as RT-CORBA lacksschedulability analysis and run-time scheduling mecha-nisms that can provide real-time guarantees to aperiodictasks. This paper makes the following contributions to thestate of the art for end-to-end aperiodic scheduling in mid-dleware. First, we compare two approaches to aperiodicscheduling, the deferrable server and the aperiodic utiliza-tion bound, using representative workloads. Numerical re-sults show that the deferrable server analysis is less pes-simistic than the aperiodic utilization bounds when appliedoffline. Second, we propose a practical approach to tuningdeferrable servers for end-to-end tasks. Third, we describedeferrable server mechanisms we have developed for TAO'sfederated event channel. Finally, we present empirical re-sults from a Linux testbed that demonstrate the efficiency ofthose deferrable server mechanisms.

Washington University in St. Louis
School of Engineering & Applied Science

Department of Computer Science & Engineering

2005-57

End-to-End Scheduling Strategies for Aperiodic Tasks in Middleware

Authors: Yuanfang Zhang, Chenyang Lu, Christopher Gill, Patrick Lardieri, Gautam Thaker

Corresponding Author: yfzhang@cse.wustl.edu

Abstract: Many mission-critical distributed real-time applications must handle aperiodic tasks with hard end-to-end deadlines. Existing middleware such as RT-CORBA lacks schedulability analysis and run-time scheduling mechanisms that can provide real-time guarantees to aperiodic tasks. This paper makes the following contributions to the state of the art for end-to-end aperiodic scheduling in middleware. First, we compare two approaches to aperiodic scheduling, the deferrable server and the aperiodic utilization bound, using representative workloads. Numerical results show that the deferrable server analysis is less pessimistic than the aperiodic utilization bounds when applied offline. Second, we propose a practical approach to tuning deferrable servers for end-to-end tasks. Third, we describe deferrable server mechanisms we have developed for TAO's federated event channel. Finally, we present empirical results from a Linux testbed that demonstrate the efficiency of those deferrable server mechanisms.

Type of Report: Other

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160

End-to-End Scheduling Strategies for Aperiodic Tasks in Middleware *

Yuanfang Zhang, Chenyang Lu, and Christopher Gill
Department of Computer Science and Engineering
Washington University, St. Louis, MO, USA
{yfzhang,lu,cdgill}@cse.wustl.edu

Patrick Lardieri and Gautam Thaker
Advanced Technology Laboratories
Lockheed Martin, Cherry Hill, NJ, USA
{plardieri,gthaker}@atl.lmco.com

Abstract

Many mission-critical distributed real-time applications must handle aperiodic tasks with hard end-to-end deadlines. Existing middleware such as RT-CORBA lacks schedulability analysis and run-time scheduling mechanisms that can provide real-time guarantees to aperiodic tasks. This paper makes the following contributions to the state of the art for end-to-end aperiodic scheduling in middleware. First, we compare two approaches to aperiodic scheduling, the deferrable server and the aperiodic utilization bound, using representative workloads. Numerical results show that the deferrable server analysis is less pessimistic than the aperiodic utilization bounds when applied offline. Second, we propose a practical approach to tuning deferrable servers for end-to-end tasks. Third, we describe deferrable server mechanisms we have developed for TAO's federated event channel. Finally, we present empirical results from a Linux testbed that demonstrate the efficiency of those deferrable server mechanisms.

1 Introduction

Many distributed real-time systems must handle a mix of periodic and aperiodic tasks. Some aperiodic tasks have hard end-to-end deadlines whose assurance is critical to the system. For example, in the Total Ship Computing Environment (TSCE) envisioned by the US Navy, the system generates an aperiodic alert event when a series of periodic sensor reports meet certain threat criteria. This event must be processed on multiple processors within a hard end-to-end deadline. User inputs and sensor readings may trigger various other hard real-time aperiodic tasks with hard deadlines. A key challenge in such systems is providing hard real-time guarantees to critical aperiodic tasks.

*This work was supported in part by DARPA Adaptive and Reflective Middleware Systems (ARMS) program (contract NBCHC030140). Approved for Public Release, Distribution Unlimited.

Schedulability analysis is essential for offline certification of the real-time properties of applications. Aperiodic scheduling has been studied extensively in real-time scheduling theory. Earlier work on aperiodic servers has also incorporated aperiodic tasks into periodic scheduling frameworks [16][14][8][15][9][12][13][9][3]. More recently, Abdelzaher et al. introduced new schedulability tests based on aperiodic utilization bounds [1].

However, despite significant theoretical results on aperiodic scheduling, these results have not been adopted in the standards-based distributed real-time middleware that is increasingly being used for developing distributed real-time applications. For example, current implementations of RT-CORBA [11] do not provide any of the schedulability tests or run-time mechanisms required by aperiodic servers. As a result, those middleware frameworks are currently unsuitable for applications with hard real-time aperiodic tasks.

To address the limitations of current middleware, we are developing support for end-to-end aperiodic scheduling in middleware. Our work addresses practical issues of schedulability analysis and scheduling mechanisms for end-to-end aperiodic tasks in middleware.

- *Practical Schedulability Analysis*: We present a numerical comparison of deferrable servers and aperiodic utilization bounds, using representative workloads. We also propose a practical method for tuning deferrable servers for end-to-end tasks.
- *Scheduling Mechanisms*: We have developed and implemented the first middleware-layer mechanisms for deferrable servers, in TAO's federated event channel. Empirical results on a Linux testbed demonstrate the feasibility of supporting deferrable servers in middleware with a small amount of run time overhead.

In the rest of this paper, we first formulate the scheduling problem and review prior theoretical results on the deferrable server and the aperiodic utilization bound. We then propose our tuning methodology and compare the two scheduling approaches. We present the design and evalu-

ation of the deferrable server in middleware. We present conclusions our after reviewing other related work.

2 Problem Formulation

In this section, we first describe our task model, and then provide a formulation of the scheduling problem.

2.1 Task Model

We consider a distributed system composed of m aperiodic tasks and n periodic tasks executing on s processors. Task T_i, is composed of a chain of n_i subtasks T_{ij}, $(1 \leq j \leq n_i)$, that are allocated to multiple processors. We specify the processors on which the n_i subtasks in task T_i execute by the visitation sequence $V_i = (V_{i1}, V_{i2}, ..., V_{in_i})$ of the task. The execution time for each subtask T_{ij} is C_{ij}. Due to precedence constraints, subtask $T_{i,j+1}$ cannot be released until subtask $T_{i,j+1}$ is completed. The first subtask T_{i1} of task T_i is released to processor $V_{i,1}$ at time A_i and its last subtask must complete on processor V_{i,n_i} by $A_i + D_i$ where D_i is the (end-to-end) hard deadline.

In the rest of this paper, we distinguish aperiodic tasks from periodic tasks with superscripts ^{a} and ^{p}. For instance, T_i^a $(1 \leq i \leq m)$ refers to an aperiodic task, and T_i^p $(1 \leq i \leq n)$. Each periodic task T_i^p is released periodically with a period P_i. Each aperiodic task T_i^a is released only once.

2.2 Problem Formulation

In this paper, we focus on the problem of certifying the real-time properties of a system *offline*. The goal of the schedulability analysis is to provide schedulability guarantees for a specific set of aperiodic and periodic tasks subject to their arrival patterns. In this paper, we are particularly interested in scenarios when all periodic and aperiodic tasks arrive *simultaneously*. For example, the certification process may prove that a system can meet all the deadlines when n aperiodic alerts are triggered simultaneously while m periodic tasks are already running in the system. Note that, while the aperiodic tasks are assumed to be able to arrive simultaneously, their arrival time is not assumed to be known. Therefore, the schedulability analysis must consider the worst-case arrival time of the aperiodic tasks.

3 Scheduling Strategies

We now give a brief review of two existing approaches to aperiodic scheduling: the aperiodic utilization bound [1] and the deferrable server [16]. Implementing bandwidth-preserving servers without kernel-level support for CPU reservations is challenging. We choose the deferrable server instead of the other bandwidth-preserving servers because it is more amenable to middleware-level implementation. Section 6.2 gives a detailed discussion of these issues.

3.1 Aperiodic Utilization Bound

According to the aperiodic utilization bound (AUB) [1] analysis, the system achieves the highest schedulable utilization bound under the Deadline Monotonic Scheduling (DMS) algorithm. Under DMS, a task has a higher priority if it has a shorter (end-to-end) deadline. The subtasks of a given task are synchronized by a greedy synchronization protocol, i.e., a subtask is released as soon as its predecessor finishes. Note that AUB does not distinguish aperiodic from periodic tasks. All tasks are scheduled using the same scheduling policy. In the AUB analysis, the set of *current* tasks $S(t)$ at any time t is defined as the set of tasks that have arrived but whose deadlines have not expired. Hence, $S(t) = \{T_i^a | A_i \leq t < A_i + D_i^a\} \cup \{T_i^p | A_i \leq t < A_i + D_i^p\}$. The synthetic utilization, $U_j(t)$ of processor j, is defined as $\sum_{T_i^a \in S(t)} C_{ij}^a / D_i^a + \sum_{T_i^p \in S(t)} C_{ij}^p / D_i^p$, which is the sum of individual subtask utilizations on this processor accrued over all current tasks. Under DMS, if for each task the following condition is satisfied, then provably [1], all task deadlines are met if the following condition holds:

$$\sum_{j=1}^{n_i} \frac{U_{V_{ij}}(1 - U_{V_{ij}}/2)}{1 - U_{V_{ij}}} \leq 1 \tag{1}$$

where V_{ij} is the jth processor that task T_i visits.

In the special case when the set of current tasks includes one task that visits all s processors in the system, condition (1) for that particular task dominates all conditions for all tasks in $S(t)$. We then only need to calculate $\sum_{j=1}^{s} \frac{U_j(1-U_j/2)}{1-U_j} \leq 1$ to verify the schedulability of all tasks in $S(t)$.

3.2 Deferrable Server

In the deferrable server (DS) [16] approach, a periodic subtask called the *deferrable server* is responsible for executing all aperiodic subtasks on a same processor. The deferrable server has a *budget*, B_j^s, and a period P_j^s. The budget B_j^s is replenished at the beginning of each period. The budget decreases whenever it is executing an aperiodic subtask, and it is preserved when the deferrable server is idle. The deferrable server can execute an aperiodic subtask as long as its budget has not been exhausted. All aperiodic subtasks are executed in Earliest Deadline First (EDF) order based on their end-to-end deadlines. All periodic subtasks are assigned priorities based on the DMS policy, while the deferrable server is assigned a higher priority than any other

periodic task. The subtasks of a given periodic task are synchronized by a non-greedy synchronization protocol such as the release guard [17] . The subtasks of a given aperiodic task are synchronized by a greedy synchronization protocol since they do not need to enforce any inter-arrival time.

In the rest of this subsection, we first describe the schedulability analysis for periodic tasks, followed by the schedulability analysis for aperiodic tasks.

3.2.1 Schedulability Analysis for Periodic Tasks

We apply the time demand method to determine whether all periodic tasks remain schedulable in the presence of deferrable servers according to [10]. For a job of periodic subtask T_{ij} released at a critical instant t_0, we add the amount $min\{\sum_{i=1}^{m} C_{ij}^a, B_j^s(1+\lceil\frac{R_{ij}-B_j^s}{P_j^s}\rceil)\}$ of processor time demanded by the deferrable server in the interval $[t_0, t_0+R_{ij}]$. Hence, the response time R_{ij} of subtask T_{ij} is given by:

$$R_{ij} = min\{\sum_{i=1}^{m} C_{ij}^a, B_j^s(1+\lceil\frac{R_{ij}-B_j^s}{P_j^s}\rceil)\} + \sum_{k=1}^{i} C_{ij}^p\lceil\frac{R_{ij}}{P_k}\rceil (B_j^s \le R_{ij} \le P_i) \quad (2)$$

$\sum_{k=1}^{i} C_{ij}^p\lceil\frac{R_{ij}}{P_k}\rceil$ represents the interference from all periodic subtasks with priority no lower than T_{ij} assuming they are sorted in non-decreasing order according to their deadlines.

Since a non-greedy synchronization protocol is used to synchronize the release of aperiodic subtasks, the end-to-end response time of an periodic task T_i is the sum of the response times of all its subtasks T_{ij} on different processors $\sum_{j=1}^{n_i} R_{ij}$. If $\sum_{j=1}^{n_i} R_{ij} \le D_i$, then periodic task T_i is schedulable.

3.2.2 Schedulability Analysis for Aperiodic Tasks

We adapt the analysis proposed in [2] to the case when all aperiodic subtasks on processor P_j arrive simultaneously. The worst-case response time for the aperiodic tasks occurs when they arrive at the time instant when the DS budget in the current period has just been exhausted, so that the initial delay of aperiodic requests is given by $P_j^s - B_j^s$. Then, since $k * B_j^s = \lfloor\frac{\sum_{k=1}^{i} C_{kj}^a}{B_j^s}\rfloor B_j^s$ is the total budget consumed by aperiodic subtasks whose absolute deadlines are no later than D_{ij} in k DS periods, the residual execution to be done in the next DS period is $R_k = \sum_{k=1}^{i} C_{kj}^a - k * B_j^s$. After substituting the initial delay, the consumed DS periods and the residual execution into the aperiodic guarantee condition in [2], we derive the following condition (3). This formula can still apply to *online* schedulability analysis when an aperiodic request arrives:

$$\lfloor\frac{\sum_{k=1}^{i} C_{kj}^a}{B_j^s}\rfloor P_j^s + (P_j^s - B_j^s) + (\sum_{k=1}^{i} C_{kj}^a - \lfloor\frac{\sum_{k=1}^{i} C_{kj}^a}{B_j^s}\rfloor B_j^s) \le D_{ij} \quad (3)$$

If we assign the subdeadlines such that $\sum_{j=1}^{n_i} D_{ij} = D_i$, then an aperiodic task T_i will meet its deadline D_i if (3) holds.

Since (3) is pessimistic when B_j^s can divide $\sum_{k=1}^{i} C_{kj}^a$, we make a small modification to (3) for this special case. If B_j^s can divide $\sum_{k=1}^{i} C_{kj}^a$, the schedulability test becomes:

$$(\frac{\sum_{k=1}^{i} C_{kj}^a}{B_j^s} - 1)P_j^s + (P_j^s - B_j^s) + B_j^s \le D_{ij} \quad (4)$$

since the residual execution to be done in the last period is exactly $R_k = B_j^s$.

Blocking Time Due to Nonpreemption: Formulas (3) and (4) assume all aperiodic tasks are scheduled by a preemptive EDF scheduling policy. However, in practice it is difficult to implement that feature in middleware without kernel-level support. Therefore, all aperiodic tasks are scheduled by a nonpreemptive EDF scheduling policy in our implementation. A shorter deadline aperiodic task that becomes ready when a nonpreemptive aperiodic task with a longer deadline is executing, is blocked until that longer deadline task completes. Consequently, when we want to determine whether an aperiodic task can meet all its deadlines, we must consider not only all the aperiodic tasks that have shorter deadlines than it, but also the maximum execution time among all lower priority aperiodic tasks on that processor. Let b_{ij} denote the longest time that aperiodic subtask T_{ij} can be blocked on processor j. Then, our necessary and sufficient schedulability formulas derived from (3) and (4) for the task T_{ij}, including its blocking time b_{ij}, are given by:

$$\lfloor\frac{\sum_{k=1}^{i} C_{kj}^a + b_{ij}}{B_j^s}\rfloor P_j^s + (P_j^s - B_j^s) + (\sum_{k=1}^{i} C_{kj}^a + b_{ij} - \lfloor\frac{\sum_{k=1}^{i} C_{kj}^a + b_{ij}}{B_j^s}\rfloor B_j^s) \le D_{ij} \quad (5)$$

and

$$(\frac{\sum_{k=1}^{i} C_{kj}^a + b_{ij}}{B_j^s} - 1)P_j^s + (P_j^s - B_j^s) + B_j^s \le D_{ij} \quad (6)$$

4 Heuristics for Tuning Deferrable Server

The choice of budgets and periods for deferrable servers has a significant impact on the schedulability of aperiodic and periodic tasks. The primary goal of the tuning process is to meet the deadlines of all aperiodic tasks. Subject to the schedulability of aperiodic tasks, it should also minimize the utilization of the deferrable server in order to improve the schedulability of periodic tasks. Clearly, exhaustive search of all possible configurations is impractical due to its exponential time complexity. In this section, we therefore propose a set of efficient heuristics for tuning deferrable servers.

4.1 Overview of the Tuning Method

Our method for tuning deferrable servers works as follows. (1) We first assign a subdeadline to each aperiodic subtask. The sum of the subdeadlines of all the subtasks of a task equals its end-to-end deadline. We give two simple subdeadline assignment policies in Section 4.2. (2) We then select a budget for each deferrable server based on the execution times of aperiodic subtasks on the same processor. We propose heuristics for selecting budgets in Section 4.3. (3) Given the subdeadlines of aperiodic subtasks and the budgets of deferrable servers, we then compute the maximum period for each deferrable server that can guarantee the aperiodic subdeadlines on the same processor based on the schedulability analysis described in Section 3.2.2.

Note that the method above guarantees the subdeadlines of all aperiodic subtasks, because the periods of deferrable servers are computed based on the schedulable condition for aperiodic subtasks, and therefore the deadlines of all tasks are guaranteed. After the parameters of the deferrable servers have been decided, the final step is testing the schedulability of all periodic tasks in the presence of deferrable servers with the selected budgets and periods using formula (2).

4.2 Subdeadline Assignment

We use two simple but efficient algorithms to assign subdeadlines for each aperiodic subtask. Similar heuristics have been used in subdeadline assignment for periodic tasks [7].

Even Deadline (ED): Evenly divide the deadline among all subtasks, so the basic rule is $D_{ij} = \frac{D_i}{n_i}$. Since all aperiodic tasks are scheduled by a nonpreemptive EDF scheduling policy in our implementation, according to formula (5), the subdeadline of T^a_{ij} should be longer than the total execution time of aperiodic subtasks on that processor whose priorities are no lower than T^a_{ij} plus the possible blocking time b_{ij} of T^a_{ij}. This slight change may make T^a_{ij}, which can not be scheduled under the basic rule, schedulable:

$$D_{ij} = \begin{cases} \frac{D_i}{n_i} & \text{if} \frac{D_i}{n_i} \geq \sum_{k=1}^{i} C^a_{ij} + b_{ij} \\ \sum_{k=1}^{i} C^a_{ij} + b_{ij} & \text{otherwise} \end{cases}$$

Proportional Deadline (PD): Make the subdeadline proportional to the execution time. We make the same change as in ED for the basic rule $D_{ij} = D_i \frac{C^a_{ij}}{\sum_{k=1}^{n_i} C^a_{ik}}$ by adding another case to it as well:

$$D_{ij} = \begin{cases} D_i \frac{C^a_{ij}}{\sum_{k=1}^{n_i} C^a_{ik}} & \text{if} \frac{D_i C^a_{ij}}{\sum_{k=1}^{n_i} C_{ik}} \geq \sum_{k=1}^{i} C^a_{ij} + b_{ij} \\ \sum_{k=1}^{i} C^a_{ij} + b_{ij} & \text{otherwise} \end{cases}$$

4.3 Budget Selection

To benefit the schedulability of periodic tasks while guaranteeing the schedulability of all aperiodic tasks in the system, we try to find the proper budget and period which make the DS task's utilization on each processor reach its lower bound. We give B^s_j/P^s_j as the utilization of the DS on processor j.

THEOREM 3.3. The utilization of the DS reaches its lower bound $max\{(\sum_{k=1}^{i} C^a_{kj} + b_{ij})/D_{ij}\}$ on processor j when DS budget B^s_j can divide $\sum_{k=1}^{i} C^a_{kj} + b_{ij}$ for any i $(1 \leq i \leq m)$.

Proof As Described. From Section 3, when $\sum_{k=1}^{i} C^a_{kj} + b_{ij}$ is not divisible by the DS budget B^s_j, the necessary and sufficient schedulability condition is inequality (5). The formula can be rewritten as:

$$\lfloor \frac{\sum_{k=1}^{i} C^a_{kj} + b_{ij}}{B^s_j} \rfloor + 1 \leq \frac{D_{ij} - \sum_{k=1}^{i} C^a_{kj} - b_{ij}}{P^s_j - B^s_j} \quad (7)$$

noting that:

$$\lfloor \frac{\sum_{k=1}^{i} C^a_{kj} + b_{ij}}{B^s_j} \rfloor + 1 > \frac{\sum_{k=1}^{i} C^a_{kj} + b_{ij}}{B^s_j}$$

Substituting the above inequality's left part with its right part into inequality (7) we obtain:

$$B^s_j/P^s_j > (\sum_{k=1}^{i} C^a_{kj} + b_{ij})/D_{ij} \quad (8)$$

When $\sum_{k=1}^{i} C^a_{kj} + b_{ij}$ is divisible by the DS budget B^s_j, the necessary and sufficient schedulability condition is inequality (6). the formula can be rewritten as:

$$\frac{\sum_{k=1}^{i} C_{kj}^{a} + b_{ij}}{B_j^s} P_j^s \leq D_{ij}$$

Thus, the DS's utilization is:

$$B_j^s / P_j^s \geq (\sum_{k=1}^{i} C_{kj}^{a} + b_{ij}) / D_{ij} \quad (9)$$

From (8) and (9), the minimum DS utilization on processor j is $max\{(\sum_{k=1}^{i} C_{kj}^{a} + b_{ij}) / D_{ij}\}$ when $\sum_{k=1}^{i} C_{kj}^{a} + b_{ij}$ is divisible by B_j^s, for any i ($1 \leq i \leq m$). This bound is tight. When $\sum_{k=1}^{i} C_{kj}^{a} + b_{ij}$, for any i ($1 \leq i \leq m$), is divisible by B_j^s, we can choose a proper P_j^s to achieve this lower bound. When not divisible, the DS utilization is always higher than the bound.

Due to the above proof, the DS budget should be a common divisor (CD) of $\sum_{k=1}^{i} C_{kj}^{a} + b_{ij}$ for each aperiodic subtask T_{ij} ($1 \leq i \leq m$) on processor P_j. This will reduce the required utilization of the Deferrable Server task, and thus enhance the schedulability of periodic tasks on that processor. Although we decide to pick the common divisor as the DS budget, there is still another question of how to choose from multiple common divisors. If a larger common divisor is picked, it will extend the interference introduced by the DS when lower priority periodic tasks are released with aperiodic tasks at the beginning of the DS period. The response times of periodic tasks thus may be increased by this longer interference on that processor. This negative impact finally may affect the schedulability of periodic tasks with shorter deadlines. If we pick a smaller common divisor, it will give us better schedulability analysis results, but may increase the run-time overhead. We compare the schedulable capabilities of these two choices in Section 5.

Although in theory 1 is always the minimum common divisor in whatever unit of time is being considered, a system may be able to *measure* time at a much finer granularity (for example in nanoseconds on a GHz processor) than it can efficiently *schedule* operations. It is therefore most appropriate to consider the smallest common divisor relative to a minimum enforcable scheduling interval, which we call the *smallest relative common divisor*. For middleware scheduling [5], 10 ms (capable of scheduling periodic tasks at rates up to 100 Hz) is a reasonable and efficiently enforcable minimum interval. The worst case occurs when we can not find a smallest relative common divisor for the set $\sum_{k=1}^{i} C_{kj}^{a} + b_{ij}$ for any i ($1 \leq i \leq m$). If this happens, we round up each $\sum_{k=1}^{i} C_{kj}^{a} + b_{ij}$ in the set to the next highest multiple of 10 ms, so that the smallest relative common divisor is always 10 ms, which is also the smallest acceptable DS budget in our system.

5 Numerical Comparison

We now compare different end-to-end scheduling strategies for aperiodic and periodic tasks through numerical analysis. The system settings used in our analysis are close to those for a representative application defined in DARPA's Adaptive and Reflective Middleware Systems (ARMS) program.

The system has 4 processors and 12 tasks including 4 aperiodic tasks and 8 periodic tasks. The number of subtasks per task is uniformly distributed between 1 and 4. Subtasks are randomly assigned to processors as long as any two subtasks in a same task are not assigned to the same processor. The periods of periodic tasks are randomly chosen between 250 ms and 10sec.

All processors have the same synthetic utilization if all subtasks are released simultaneously. We vary the synthetic utilization per processor in different runs to evaluate the impact of system load on the schedulability analyses. 100 different task sets were randomly generated for each synthetic utilization. As is described in Section 3, we consider the worst-case scenario in which all aperiodic and periodic tasks are released simultaneously.

In the previous section we proposed two heuristics for selecting budgets for deferrable servers: the *smallest relative common divisor* (SRCD) and the *greatest common divisor* (GCD). We also have two subdeadline assignment heuristics: Proportional Deadlines (PD) and Even Deadlines (ED). The combinations of the heuristics produce 4 different DS configurations referred to as DS+ED+SRCD, DS+ED+GCD, DS+PD+SRCD and DS+PD+GCD. We compare our heuristics with exhaustive search (ES) which tries to search all combinations of possible subdeadline assignments and budget selections. Since the time complexity of exhaustive search is exponential, we only consider 6 possible values for each subdeadline and budget. The range for each budget of a deferrable server is from 10 ms to 25 ms. The subdeadline range of one subtask is from its execution time to its longest allowed subdeadline. For example, if Task T_i has 4 subtasks, assume the subdeadlines of the first 2 subtasks have been decided. Then the longest allowed subdeadline for the third subtask is equal to $D_i - \sum_{j=1}^{2} D_{ij} - C_{i4}$. The schedulability analysis with DS+ES still takes several hours to run even when we limit it to this 6-way search.

We plot the fraction of task sets that are schedulable under different analyses in Figure 1. We first compare the results under different variations of the DS approach. As expected based on the earlier discussion, SRCD achieves higher schedulability than GCD, while the subdeadline assignment heuristics do not have a significant effect on schedulability. Furthermore, SRCD performs close to ES, indicating our efficient heuristics for tuning de-

ferrable servers can be highly effective in practice.

Our results also show that the AUB analysis is more pessimistic than the DS analysis when they are both performed *offline*. However, earlier results demonstrated that the AUB approach may significantly reduce its pessimism when it is applied *online* because it can reset the synthetic utilization to zero whenever the processor becomes idle. Therefore our results may only hold in the offline case which is the focus of this study.

To be more general, we increase the number of the processors in the system to 8. At the same time, the number of subtasks per task is uniformly distributed between 1 and 8. Keeping other parameters the same, we randomly generate same number of task sets and do schedulability analysis for each of them. Since the running time of exhaustive search increases exponentially, when we increase the number of processors and the maximum number of subtasks per task, we only compare our heuristics with AUB as is shown in Figure 2. Although increasing the number of subtasks and processors decreases the schedulable task sets for both approaches, our heuristics still show better schedulability results than AUB when performed offline.

6 Deferrable Server in Middleware

To support aperiodic tasks on middleware, we have integrated deferrable servers with the event service of The ACE ORB (TAO) [6]. In this section, we first give an overview of TAO's federated event channel [6], and then present the design and implementation of deferrable server mechanisms in that event channel.

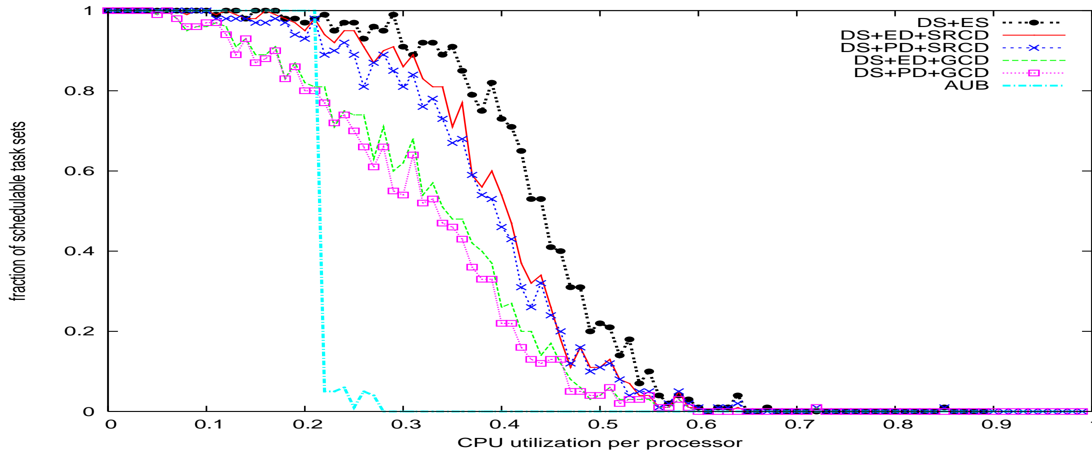

Figure 1. Schedulability Comparison for 4 processors with Nonpreemptive EDF

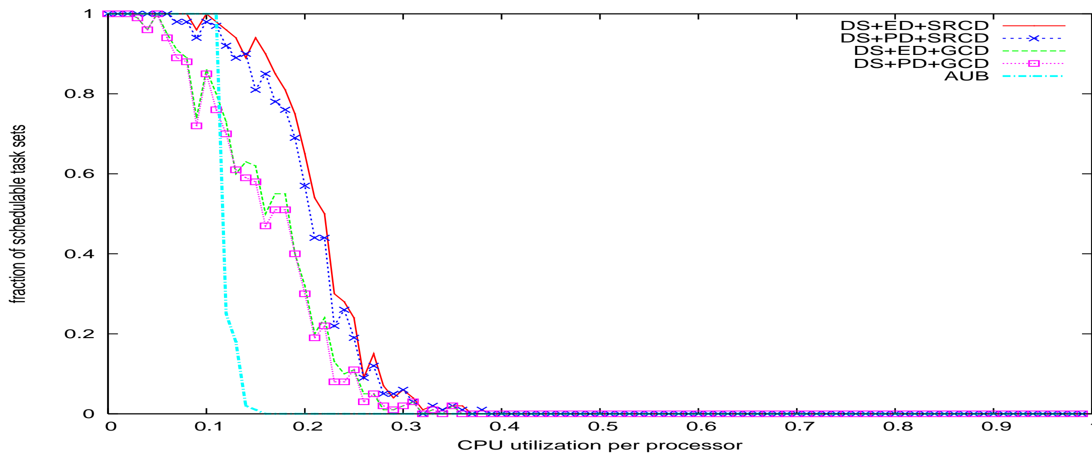

Figure 2. Schedulability Comparison for 8 processors with Nonpreemptive EDF

6.1 TAO's Federated Event Channel

Each processor has its own event channel (EC), and the ECs exchange events via a Gateway, as described in [6]. The Gateway can reside in either processor, but for our implementation it is on the supplier side. Each subtask is implemented as a supplier-consumer pair. The supplier pushes events which trigger subtask execution in a single consumer.

For example, consider a periodic task T_1^p as described in Section 3, which has 3 subtasks executing on 3 different processors in sequence. The structure of timers, suppliers, and consumers on processors i, j, and k is shown in Figure 3. In processor i, a Supplier1_1 with a timer (depicted as a clock) pushes events (arrows) through a local EC to a Consumer1_1, which then executes subtask $T_{1,1}^p$ and pushes another event to trigger execution on processor j through another Supplier1_2, local EC, Gateway and remote EC. Processor j's Consumer1_2 executes subtask $T_{1,2}^p$ and then pushes yet a third event to trigger execution on processor k, whose Consumer1_3 executes the final subtask, $T_{1,3}^p$. This sequence of events happens periodically for a periodic task, every time the timer determines that it is time for T_1^p to release a job. For an aperiodic task T_1^a which also has 3 subtasks executing on processors i, j and k, its sequence of events is the same as the periodic task, except that its sequence of events happens only once.

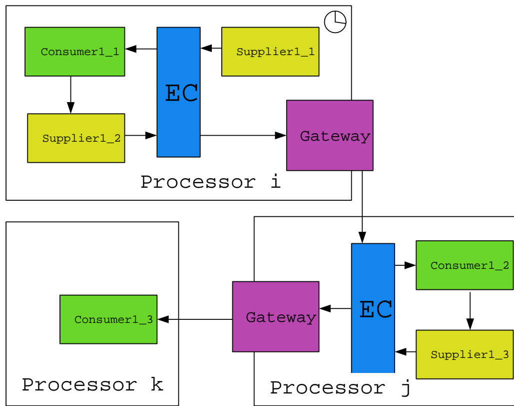

Figure 3. Federated Event Channel Structure

The Kokyu Dispatching framework [5] is used in our system to provide real-time dispatching of events. Release Guard was implemented in [18] for ECs in TAO as a slight modification of the Kokyu Dispatching Framework. We configure Kokyu to use the preemptive DMS scheduling algorithm. Each periodic event is assigned to a specific dispatching queue according to its period. Each dispatching queue has an operating system thread whose priority is decided by the queue priority. Each dispatching thread removes the event from the head of its queue and runs its entry point function until it completes or is preempted by a higher priority dispatching thread.

6.2 Deferrable Server Implementation

The original Kokyu dispatching framework [5] does not support any bandwidth-preserving servers for aperiodic events. As we noted in Section 3, it is challenging to implement bandwidth preserving servers such as the deferrable server on top of standard operating systems (e.g., Linux) that do not support CPU reservation.

We overcome this challenge by developing a novel deferrable server mechanism in TAO. Our deferrable server is implemented by a pair of threads: a *deferrable server* thread that processes aperiodic events and a *budget manager* thread that manages the budget and the execution of the deferrable server thread. The deferrable server thread, which is assigned the second highest operating-system thread priority, services all aperiodic tasks. If an aperiodic event arrives and the deferrable server thread does not use up its budget in the current period, the aperiodic task will be able to preempt any other running periodic task. The budget manager thread runs at the highest operating system priority to manipulate the consumption and replenishment of the DS's budget. Its highest priority allows it to interrupt the DS thread when it uses up its budget in the current period. The two threads share two variables, *left-exec* and *left-budget*. *left-exec* keeps track of the remaining execution time for the currently running aperiodic task, and *left-budget* keeps track of the remaining budget in the current period.

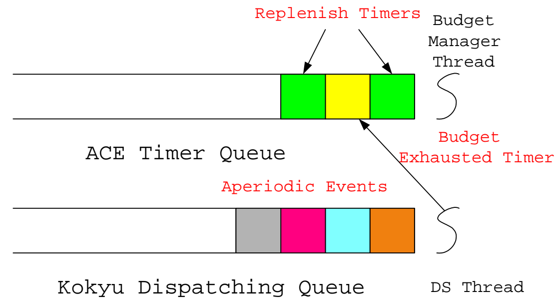

Figure 4. Middleware DS Mechanisms

As we show in Figure 4, the deferrable server thread dispatches all aperiodic events. Before dispatching an event, if it finds there is not enough budget left for the aperiodic event to complete in the **current** period, it will insert a *budget exhausted* timer into a timer queue. The *budget exhausted* timer will fire when the remaining budget has been used up, and generate a *budget exhausted* event. This budget exhausted event is handled by the budget manager thread which runs at the highest priority. When this event happens, the budget manager thread will send a suspend signal

to the deferrable server thread.

The budget manager thread also handles a timer-driven event called *replenish*, which occurs at the beginning of each DS period. When this event happens, the budget manager thread will replenish the DS budget and will also send a resume signal to the DS thread if it had been suspended previously. It may set a new *budget exhausted* timer which will fire when the replenished budget has been used up, if it finds the remaining execution time of the current running aperiodic task is longer than the replenished budget.

We summarize the operations involved in our deferrable server mechanism as follows:

Budget manager: On a periodic *replenish* event, (1) reset *left-budget* and update *left-exec*, (2) resume the deferrable server thread if it is suspended, and (3) insert a *budget exhausted* timer into the timer queue if *left-exec* is larger than the replenished budget. On a *budget exhausted* event, simply (1) suspend the deferrable server thread.

DS thread: before dispatching an aperiodic task, (1) update *left-budget* and *left-exec*, and (2) insert a *budget exhausted* timer into the timer queue if *left-budget* is less than the task's execution time.

We note that our budget management mechanism will function properly only if the DS thread runs at a higher priority than the other dispatching threads, and can only be interrupted by the budget manager thread.

7 Empirical Evaluation

7.1 Experimental Platforms

The experiments described in this section were performed using ACE/TAO version 5.1.4/1.1.4 on a testbed consisting of four machines. Two of them are Pentium-IV 2.5GHz machines, and the other two are Pentium-IV 2.8GHz machines. Each of them has 500MB RAM and 512KB cache, and runs version 2.4.22 of the KURT-Linux operating system [4]. These platforms provide a CPU-supported timestamp counter with nanosecond resolution.

The data stream user interface (DSUI) and data stream kernel interface (DSKI) are provided with the KURT-Linux distribution. The DSUI is used to record information from the middleware and application layers, while the DSKI is used to collect information at the kernel level like context switching. By using both DSUI and DSKI instrumentation, we can obtain a precise accounting of task start and stop times, thread context switches, CPU idle intervals, and other relevant events across multiple system levels.

7.2 Schedulability Validation

For CPU utilizations 0.4 and 0.5, we randomly picked 20 task sets. Half of them are schedulable in DS+ED+SRCD with nonpreemptive EDF, and half are not. We ran each task set for 10 minutes and checked how many task sets had deadline misses in our DS implementation. We did the same thing for CPU utilizations 0.6, 0.7, 0.8 and 0.9. For CPU utilization 0.6, only DS+ED+SRCD and ES could find a schedulable task set out of 100 randomly generated task sets, and we also randomly picked 10 unschedulable sets at that CPU utilization. We only randomly picked 10 unschedulable sets for each of utilizations 0.7, 0.8 and 0.9, since at those CPU utilizations, none of the analyses can find any schedulable task sets among our 100 randomly generated task sets.

Our results show that no schedulable task sets had deadline misses in a run of 10 minutes. Some of the analytically unschedulable task sets also met all deadlines in our experiments. This is because the worst-case arrival pattern (often referred to as the *critical instant*) assumed by the analysis did not occur in our experiments. We note that it is impossible to synchronize the release times of all subtasks on all processors due to the dependencies among the subtasks.

To measure the pessimism of the analysis, we also plot the fraction of the analytically unschedulable task sets that also missed deadlines in our experiments in Figure 5. The fraction of task sets with missed deadlines increased with increasing utilization, indicating that the analysis is less pessimistic under heavy load.

Moreover, for some CPU utilizations, DMS can satisfy all deadlines of the task set in which DS has deadline misses. This result indicates that, for our workload, DMS can meet more deadlines than DS *in practice* even though the AUB *analysis* is more pessimistic than the Deferrable Server analysis.

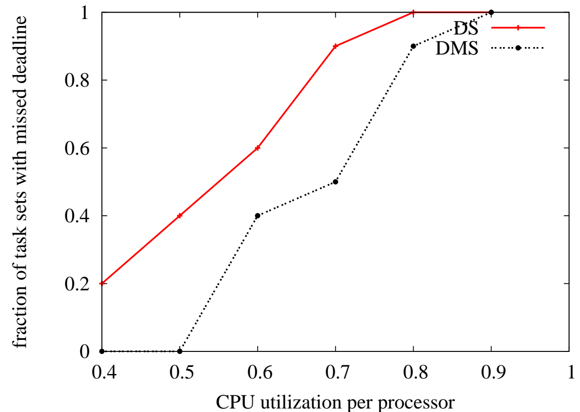

Figure 5. Fractions of analytically unschedulable task sets that actually missed deadlines

7.3 Overhead of Deferrable Servers

To measure the run-time overhead incurred by deferrable servers, we first randomly generated one task set when the CPU utilization was 0.5, then used the DS+ED+SRCD strategy to calculate the DS's budget and period. After running the task set for 5 minutes, we collected the data recorded by DSUI and DSKI, and then calculated the overhead.

BM: replenish	(1) reset	(2) resume DS	(3) set timer
	10.010 μs	5.606 μs	9.775 μs
BM: budget exhausted	(1) suspend DS		
	7.003 μs		
DS thread	(1) update	(2) set timer	
	15.729 μs	30.186 μs	

Table 1. Deferrable Server Overhead Analysis

In each period of the deferrable server, the budget manager thread is released at most twice. It is released in the beginning of each period to handle the *replenish* event, and it may be released again to suspend the execution of the deferrable server thread when the *budget exhausted* event happens. Since not all jobs are necessary for each release, we measure the execution time of each step and average the results from a 5 minute run, shown in Table 1.

The maximum overhead for each operation is the sum of the overheads of all its steps. These overheads are small when compared with the execution time of tasks in our system. Since in each DS period, the replenish operation runs once and the exhausted operation runs at most once, the maximum budget manager overhead per DS period is the sum of the maximum overheads of its replenish and budget exhausted operations. Our measurements showed this was as high as 32.394 μs. Moreover, some steps may not be invoked for some releases. For example, in most cases, steps 2 and 3 of the replenish operation are not invoked because their conditions are not satisfied. Step 2 runs only if the DS thread has been suspended because of the *budget exhausted* event. Step 3 runs only if the remaining execution time of the current running aperiodic task is longer than the replenished budget. So their conditions can only be satisfied when aperiodic tasks are running. Since each aperiodic task only releases once, the proportional overhead, when an aperiodic task is running, is small over a run of 5 minutes. We also measure the average overhead of the replenish operation by averaging total overhead over the total number of releases in 5 minutes. For the replenish operation, its average overhead for each release is about 7.896 μs, which is close to the average overhead of its first step.

The overhead for the DS thread's task dispatching operation, since it only happens when we release an aperiodic task and our workload only has 4 aperiodic tasks, is at most 4 times the sum of the overheads of its steps 1 and 2, per processor. Our measurements showed that this was as high as 183.66 μs. All of these results demonstrate that deferrable servers can be supported at the middleware layer with acceptable run-time overhead.

In addition, we found that step 3 of the replenish operation and step 2 of the DS thread operation, both of which insert a *budget exhausted* timer into the Timer Queue, have very different overheads. Looking at the detailed timeline data collected by DSUI and DSKI, we found several context switches between the DS thread and the Budget Manager thread when the DS thread inserts a timer into the timer queue, which do not occur when the replenish operation similarly inserts a timer. The DS budget manager thread is always waiting on a reactor timer, and will be sitting in a `select` system call when the DS thread makes a call to set a new timer. If the new timer is shorter than the existing one, the select call must be triggered to return early via the reactor's notification pipe, so that the new earlier wait time can be used instead of the old later time. The budget manager thread is thus being woken up and run briefly when the DS thread tries to insert a timer into that Timer Queue, to change to the new earlier time and to use that value in a new call to select, on behalf of the DS thread.

8 Other Related Work

In addition to the Aperiodic Utilization Bound (AUB) [1] and the Deferrable Server (DS) [16] algorithms we considered in this paper, there are many other scheduling algorithms that deal with aperiodic and periodic hybrid task sets. The simplest method to handle a set of soft aperiodic tasks in the presence of periodic tasks is to schedule them in the background. Since aperiodic tasks' execution can be interrupted by any periodic task while in background processing, hard deadlines of aperiodic tasks in our system may not be satisfied. Another scheduling algorithm which is similar to Deferrable Server is Polling Server (PS) [14]. The major difference between DS and PS is that the PS does not preserve its budget. If no aperiodic tasks are pending, PS suspends itself until the beginning of its next period and its higher priority budget in this period is wasted. Neither background processing nor PS is suitable for aperiodic tasks with hard deadlines as they may suffer from long response times.

Two other algorithms which are both bandwidth-preserving servers like DS are Priority Exchange (PE) [8] and the Sporadic Server (SS) [15]. Unlike DS, PE preserves its high priority budget by exchanging it for the execution time of a lower priority periodic task. So the PE algorithm has a slightly longer aperiodic response time while bene-

fiting the schedulability of periodic tasks. Moreover, PE requires extra run-time overhead to manage and track priority exchanges when compared with DS. SS only replenishes its capacity after it has been consumed by aperiodic tasks. Compared with DS and PS, the Sporadic Server enhances the average response time of aperiodic tasks while not affecting the schedulability of periodic tasks. With respect to DS, the budget and period of SS are not fixed. SS needs to calculate its replenishment time and amount frequently, which costs more run-time overhead.

Another well-known algorithm that supports hybrid periodic and aperiodic task set scheduling is Slack Stealing [9][12][13]. Slack Stealing tries to steal any slack from the periodic tasks and uses it to execute aperiodic tasks as soon as possible. So Slack Stealing has the shortest average aperiodic task response time over DS, PE and SS. Slack Stealing has a static version [9] and a dynamic version [3], depending on whether or not the slack is dynamic. Both versions are hard to implement in a real system, because of their space and time complexity.

9 Conclusions

In summary, this work represents a promising step toward developing integrated end-to-end scheduling services for aperiodic and periodic tasks in distributed real-time middleware. We found that the aperiodic utilization bound analysis is more conservative than the deferrable server analysis when they are used offline. Second, we proposed efficient heuristics for tuning deferrable servers for end-to-end tasks. Numerical results demonstrate that our heuristics are competitive against exhaustive search. Moreover, we have designed and implemented a novel deferrable server mechanism and integrated it with TAO's federated event channel. Our deferrable server mechanism is highly efficient, incurring less than 15 μs of average run-time overhead per period on a Linux testbed. In the future, we plan to integrate end-to-end scheduling services with online admission control and task (re)allocation in dynamic environments.

References

[1] T. F. Abdelzaher, G. Thaker, and P. Lardieri. A Feasible Region for Meeting Aperiodic End-to-end Deadlines in Resource Pipelines. In *International Conference on Distributed Computing Systems ICDCS 2004*, Tokyo, Japan, Mar. 2004.

[2] G. C. Buttazzo. *Hard Real-Time Computing Systems*. Kluwer Academic Publishers, Norwell, Massachusetts, 1997.

[3] R. I. Davis, K. Tindell, and A. Burns. Scheduling slack time in fi xed priority preemptive systems. In *IEEE Real-Time Systems Symposium*, pages 222–231. IEEE, 1993.

[4] Douglas Niehaus, *et al.*. Kansas University Real-Time (KURT) Linux. `www.ittc.ukans.edu/kurt/`, 2004.

[5] C. Gill, D. C. Schmidt, and R. Cytron. Multi-Paradigm Scheduling for Distributed Real-Time Embedded Computing. *IEEE Proceedings, Special Issue on Modeling and Design of Embedded Software*, 91(1), Jan. 2003.

[6] T. H. Harrison, D. L. Levine, and D. C. Schmidt. The design and performance of a real-time CORBA event service. In *Proceedings of OOPSLA '97, Atlanta, GA*, pages 184–200, 1997.

[7] B. Kao and H. Garcia-Molina. Deadline assignment in distributed soft real-time systems. In *Proceedings of the 13th International Conference on Distributed Computing Systems*, pages 428–437. IEEE, 1993.

[8] J. P. Lehoczky, L. Sha, and J. K. Strosnider. Enhanced aperiodic responsiveness in a hard real-time environment. In *IEEE Real-Time Systems Symposium*, pages 261–270. IEEE, 1987.

[9] J. P. Lehoczky and S. R. Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fi xed-priority preemptive systems. In *The 13th IEEE Real-Time Systems Symposium (RTSS '92)*, pages 110–123, Phoenix AZ, 1992.

[10] J. W. S. Liu. *Real-Time Systems*. Prentice Hall, New Jersey, 2000.

[11] Object Management Group. *Real-Time CORBA Specification*, 1.1 edition, Aug. 2002.

[12] S. Ramos-Thuel and J. P. Lehoczky. On-line scheduling of hard deadline aperiodic tasks in fi xed-prioriry systems. In *IEEE Real-Time Systems Symposium*, pages 160–171. IEEE, 1993.

[13] S. Ramos-Thuel and J. P. Lehoczky. Algorithms for scheduling hard aperiodic tasks in fi xed priority systems using slack stealing. In *IEEE Real-Time Systems Symposium*, pages 22–35. IEEE, 1994.

[14] L. Sha, J. P. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritizing preemptive scheduling. In *IEEE Real-Time Systems Symposium*, pages 181–191. IEEE, 1986.

[15] B. Sprunt, L. Sha, and L. Lehoczky. Aperiodic task scheduling for hard real-time systems. *The Journal of Real-Time Systems*, 1(1):27–60, 1989.

[16] J. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in real-time environments. *IEEE Transactions on Computers*, 44(1):73–91, 1995.

[17] J. Sun. *Fixed priority scheduling of end-to-end periodic tasks*. PhD thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, 1997.

[18] Yuanfang Zhang, Bryan Thrall, Stephen Torri, Christopher Gill and Chenyang Lu. A Real-Time Performance Comparison of Distributable Threads and Event Channels. In *Proceedings of the 11th Real-time Technology and Application Symposium (RTAS '05)*, San Francisco, Mar. 2005. IEEE.