1996-12-04

17th IEEE Real-Time Systems Symposium: Work in Progress Sessions

Archer, Myla

Boston University Computer Science Department


http://hdl.handle.net/2144/3747

Boston University
Dynamic Scheduling of Hard Real-Time Applications in Open System Environment

Z. Deng  J. W.-S. Liu  J. Sun
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801

1 Introduction

With tremendous advances in hardware technologies, it is now possible to run real-time applications on fast, general purpose workstations and personal computers concurrently with non-real-time applications. A challenging problem is how to schedule an open system of complex, independently developed real-time applications and non-real-time applications. A scheduling scheme for this purpose should meet the following objectives.

1. It allows the developer of each real-time application to validate the schedulability of the tasks in the application in isolation from other applications in a way appropriate for the application.

2. It has a simple acceptance test according to which the operating system can determine whether to admit a new real-time application into the system.

3. Once the system admits a real-time application, it guarantees the schedulability of individual tasks in the application.

4. It maintains a certain level of responsiveness of non-real-time applications.

5. It does the above without relying on fixed allocation of time/resources or fine-grain time slicing and consequently, is suited for applications with varying time/resource demands and stringent timing requirements.

We are developing a two-level hierarchical scheme that meets these objectives. It assumes that when the operating system admits a new real-time application into the system, it creates a dedicated constant utilization server (which we will describe shortly) to execute the application. All non-real-time applications are executed by one server. At the top level, the operating system allocates processor time to the servers and schedules the servers according to the earliest-deadline-first (EDF) algorithm. At the low level, the tasks in each application are scheduled by their server scheduler according to an algorithm chosen for the application. The scheduling algorithm for each real-time application can be either preemptive or nonpreemptive. We have shown that the schedulability of any application containing arbitrary tasks can be validated independently of other applications if the application uses a nonpreemptive priority-driven scheduling algorithm. Similarly, we have shown that any application can be validated independently if it uses a preemptive priority-driven scheduling algorithm, provided that the release time of all its jobs are known (e.g., in the case of periodic tasks). Overall, non-real-time applications are scheduled in a time-sharing fashion.

2 Scheduling in Open System

Figure 1 shows our open system. The system has a processor with speed equal to one. The workload on the processor consists of real-time applications, denoted by $A_k, k = 1, 2, \ldots$ and so on, and non-real-time applications. We assume that the applications are independent of each other and that every real-time application $A_k$ is schedulable when it executes alone on a slow processor with speed $\sigma_k < 1$. Each real-time application $A_k$ is executed by a dedicated server $S_k$ for $k \geq 1$, and all the non-real-time applications are executed by the server $S_0$.

2.1 Scheduling Hierarchy

The applications are scheduled and executed according to a two-level hierarchical scheme. At the top level, the operating-system scheduler maintains all the servers. (Hereafter, we refer to this scheduler as the OS scheduler.) It replenishes the server budget and sets the server deadline for every server in the system and schedules all the servers in the system according to the earliest-deadline-first (EDF) algorithm.

At any time, the system consists of a number of servers, as shown in Figure 1. Each server $S_k$ has a ready queue that contains ready-to-run jobs to be executed by the server. When the OS scheduler selects a server to execute, the server executes the job at the head of its ready queue. The server $S_k$ for each real-time application $A_k$ in the system has a low-level server scheduler, which schedules ready-to-run jobs in $A_k$ and places them in priority order in the ready queue of $S_k$. The server scheduler is a part of the application. In contrast, the OS scheduler schedules all the non-real-time applications. The net effect is that all the non-real-time applications appear to be running in a slower time-sharing environment.

More specifically, when a job of a real-time application $A_k$ is released, the operating system invokes the server scheduler of the server $S_k$. The server scheduler then inserts the newly released job in the proper location in the server's ready queue according to the scheduling algorithm used by the server scheduler. We assume that the algorithm used by every server scheduler is a simple priority-driven algorithm. The time taken for handling
the insertion of the newly released job into the ready queue is either negligibly small compared with the execution times of all the jobs in the system or is accounted for by including the server scheduler as a task of \( A_k \) when determining the schedulability of \( A_k \).

### 2.2 Constant Utilization Server

As stated earlier, all the servers in the system are Constant Utilization Servers, which are called Total Bandwidth Servers by Spuri and Buttazzo [1]. A constant utilization server \( S_k \) is defined by its server size \( U_k \), which is the fractional processor utilization allocated to the server. The server budget \( b_{k,0} \) and server deadline \( d_{k,0} \) of each server \( S_k \) are zero initially. The server budget is replenished whenever certain events of the application \( A_k \) occur; we will return shortly to describe what these events are. Let \( t_{k,i} \) denote the occurrence time of the \( i \)-th application event of \( A_k \) (i.e., the time instant when the budget of the server \( S_k \) is replenished for the \( i \)-th time.) Let \( \epsilon_{k,i} \) denote the amount of budget replenished at the time. The deadline \( d_{k,i} \) set for server is equal to \( t_{k,i} + \epsilon_{k,i}/U_k \). The server behaves like a task with a constant utilization \( U_k \) if its ready queue is never empty, thus the name Constant Utilization Server.

We say that a server is schedulable if every time after the server budget and deadline are set, its budget is always exhausted at or before the deadline. Stated in another way, we can view each server as a sporadic task in which a job with execution time equal to the server budget is released each time the server budget is replenished. The server is schedulable when every job of it completes by its deadline. We have shown that a system of constant utilization servers is schedulable according to the EDF algorithm if their total size is equal to or less than one [2].

### 3 OS Scheduler

The operation of the OS scheduler is shown in Figure 2. When the system starts, the OS scheduler creates the server \( S_0 \) for the non-real-time applications. The OS scheduler always admits non-real-time applications. A real-time application \( A_k \) that is schedulable on a slow processor with speed \( \sigma_k \) requests for admission by informing the OS scheduler its required processor bandwidth \( \sigma_k \). The OS scheduler admits the application if the total server size \( U_t \) is equal to or less than \( 1 - \sigma_k \) at the time.

When the OS scheduler admits a new real-time application \( A_k \), it creates a server \( S_k \) with server size \( \sigma_k \) to execute \( A_k \). When the application \( A_k \) is terminated, the OS scheduler destroys the server \( S_k \). Because the total server size of all servers in the system never exceeds one, all the servers are schedulable at all times.

### 4 Server Maintenance

The OS scheduler maintains the server \( S_k \) depending on whether the tasks of \( A_k \) are scheduled preemptively or nonpreemptively and whether the tasks in \( A_k \) contend for resources amongst themselves. Figure 3 describes how the server of a preemptive application \( A_k \) is maintained when the tasks in \( A_k \) do not contend for resources. In this case, we require that the release times of all jobs in the application \( A_k \) be known a priori, as in the case when all tasks are periodic.

---

**Figure 1:** Open System Model

**Figure 2:** Operations of the OS scheduler

**Initiation:**
- Create a constant utilization server \( S_0 \) with size \( U_0 \) for non-real-time applications.
- Set \( U_1 \) to \( U_0 \).

**Acceptance test and admission of \( A_k \) which is schedulable on a slow processor with speed \( \sigma_k \):**
- If \( U_1 + \sigma_k > 1 \), reject \( A_k \). Otherwise, admit \( A_k \), and
  - Create a constant utilization server \( S_k \) with size \( \sigma_k \) for \( A_k \).
  - Set server budget and server deadline \( d \) to zero.
  - Increase \( U_1 \) by \( \sigma_k \).

**Maintenance of each server \( S_k \):**
- If the application \( A_k \) is scheduled preemptively, maintain the server \( S_k \) as shown in Figure 3.
- Otherwise maintain the server as described in [2].

**Scheduling of all servers:**
- Schedule all servers on the EDF basis.

---
1. When a new job $J_t$ of $A_k$ arrives at $t$, invoke the server scheduler of $S_k$ to place $J_t$ in the proper location in $S_k$’s ready queue. If the current server deadline $d \leq t$,
   
   (a) invoke the server scheduler of $S_k$ to update the occurrence time $t_o$ of the next application event of $A_k$,
   
   (b) set the server budget to $(t_o - t)\sigma_k$, and server deadline $d$ to $t_o$.

2. At the deadline $d$ of the server $S_k$, if its ready queue is not empty,
   
   (a) invoke the server scheduler of $S_k$ to update the occurrence time $t_o$ of the next application event of $A_k$,
   
   (b) set the server budget to $(t_o - d)\sigma_k$, and server deadline $d$ to $t_o$.

3. When the application $A_k$ terminates,
   
   (a) delete $S_k$ from the system,
   
   (b) decrease $U_t$ by $\sigma_k$.

Figure 3: Maintenance of Server $S_k$ for a Preemptive Application $A_k$

Specifically, the real-time application $A_k$ consisting of $N$ tasks. The jobs in each task are released periodically. $A_k$ is schedulable by itself on a slow processor with speed $\sigma_k < 1$ by some preemptive scheduling algorithm. Each task $T_i$ in $A_k$ is characterized by its phase $r_i$ and period $p_i$; the release time of the $j$-th job of task $T_i$ is $r_i + (j - 1)p_i$ for all $j \geq 1$. We also assume that the execution time $e_m$ and relative deadline $d_m$ of every job $J_m$ become known after $J_m$ is released. However, the jobs in each task may have different execution times and relative deadlines. Hence the task need not be a periodic task.

The description in Figure 3 uses the term application event. The term refers to either the release or the completion of a job, we say an application event of $A_k$ occurs when a job in $A_k$ is released or completes. At any time $t$, the next application event of $A_k$ is the application event of $A_k$ that has the earliest possible occurrence time after $t$ if the application $A_k$ executes alone on the slow processor. Let $t'$ denote the earliest release time of any job of the application $A_k$ after $t$. Then at time $t$, the next application event of $A_k$ occurs either at $t'$, if the ready queue of server $S_k$ is empty at $t$, or at $\min\{t', t + e'_j/\sigma_k\}$, if the job $J_j$ at the head of the ready queue has remaining execution time $e'_j$. The OS scheduler replenishes the server budget of $S_k$ each time an application event of $A_k$ occurs. The amount of budget replenished depends on when the next application event of $A_k$ will occur and the server budget will be replenished again. This is the reason for the restrictive assumption that the release times of all jobs are known.

If the application $A_k$ is scheduled by some nonpreemptive algorithm, the procedure the OS scheduler uses to maintain the server $S_k$ is much simpler. In particular, it is not necessary that the release time of every job be known $a$ priori; only the execution time of every job needs to be known after it is released. The way the server of a nonpreemptive application is maintained is described in [2], which also describes how a server for any preemptive application whose tasks contend for resources is maintained.

5 Schedulability of Real-Time Applications

We have proved the following schedulability conditions of a real-time application $A_k$ in the open system when it is executed by the server $S_k$ according to the two-level scheduling hierarchy described above. The proofs of these theorems can be found in [2].

**Theorem 1:** If a real-time application $A_k$ is schedulable on a slow processor with speed $\sigma_k < 1$ by itself according to some nonpreemptive algorithm, it is also schedulable on the fast processor with speed one according to the two-level scheduling hierarchy, provided that the total size of all existing servers in the system is no more than $1 - \sigma_k$.

**Theorem 2:** If a real-time application $A_k$ that consists solely of periodic tasks is schedulable on a slow processor with speed $\sigma_k < 1$ by itself according to some preemptive algorithm, it is also schedulable on the fast processor with speed one according to the two-level scheduling hierarchy, provided that the total size of all existing servers in the system is no more than $1 - \sigma_k$.

Again, if the application uses some nonpreemptive scheduling algorithm, there is no restriction in the types of tasks in it. If the application uses some preemptive scheduling algorithm, our scheme requires that the release times of all jobs be known $a$ priori. This is the difference in the statement of these two theorems.

6 Current Work

We are currently extending this two-level scheduling algorithms for various types of real-time applications in the open system. The types include:

- preemptive real-time applications consisting of periodically released jobs with release time jitters,
- preemptive real-time applications consisting of periodic tasks and sporadic tasks,
- real-time applications with local and/or global resource contentions, and
- preemptive real-time applications with nonpreemptable sections.

We plan to build a proof-of-concept prototype run-time environment that embeds on a commonly used microkernel operating system schedulers and application system interfaces based on these algorithms.
References
