AIRS: Supporting Interactive Real-Time Applications on Multicore Platforms *

Shinpei Kato[†‡], Ragunathan (Raj) Rajkumar[†], and Yutaka Ishikawa[‡]
[†]Department of Electrical and Computer Engineering, Carnegie Mellon University
[‡]Department of Computer Science, The University of Tokyo

Abstract

*Modern real-time systems increasingly operate with multiple interactive applications. While these systems often require reliable quality of service (QoS) for the applications, even under heavy workloads, many existing CPU schedulers are not very capable of satisfying such requirements. In this paper, we design and implement an **A**dvanced **I**nteractive and **R**eal-time **S**cheduler, called **AIRS**. AIRS is aimed at supporting systems that run multiple interactive real-time applications, particularly on multicore platforms. It provides a new CPU reservation mechanism to enhance the QoS of the overall system. The reservation algorithm is based on the prior Constant Bandwidth Server (CBS) algorithm, but is more flexible and efficient, when multiple applications reserve CPU bandwidth. It also provides a new multicore scheduler to improve the absolute CPU bandwidth available for the applications to perform well. The scheduling algorithm is subject to the prior Earliest Deadline First with Window-constraint Migration (EDF-WM) algorithm, but is extended to work with the new CPU reservation mechanism. Experimental evaluation shows that AIRS delivers higher quality to simultaneous playback of multiple movies than the existing real-time scheduler. It also demonstrates that AIRS offers hard timing guarantees for randomly-generated task sets with heavy workloads.*

1 Introduction

Real-time systems fulfill a significant role in societal infrastructure, with application domains ranging from safety-critical systems (e.g., cars, aircraft, robots) to more interactive systems (e.g., consumer electronics, multimedia devices, streaming servers). The safety-critical systems are usually hard real-time systems, while the interactive systems tend to be soft real-time systems, in which timing violations decrease the quality of service (QoS) of applications but does not lead to system failure. In either case, however, timely execution is desirable.

Modern real-time systems increasingly generate heavy workloads. For instance, the state-of-the-art humanoids include multiple hard and soft real-time applications [4, 20]. Digital high-definition TVs are also expected to run simultaneous playback of multiple videos [30]. Other examples include teleconferencing, video streaming, and 3D games.

*This work is supported by the fund of Research Fellowships of the Japan Society for the Promotion of Science for Young Scientists.

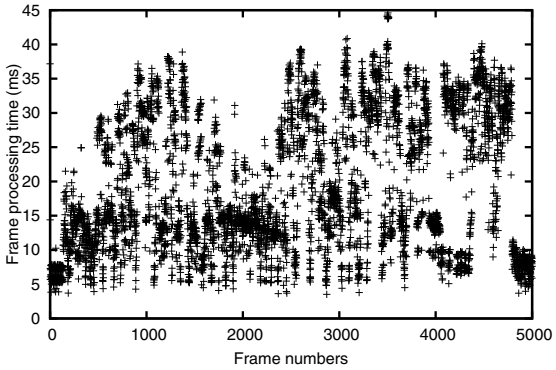


Figure 1. Frame processing times of an H264 (1920×800 at 24 fps) movie.

Many of these applications also tend to be interactive and resource-intensive. CPU schedulers in those systems are therefore required to maintain the QoS of the applications, even under heavy workloads.

CPU scheduling is a central concept in real-time operating systems (RTOSs) to satisfy the timing requirements. In particular, Earliest Deadline First (EDF) [23] is an optimal real-time scheduling algorithm for classical single-core platforms in terms of schedulability – the capability to schedule a system without any timing violations. Thus, it is applicable for heavy-workload real-time systems.

CPU reservation and multicore resource management are key techniques for CPU schedulers to further support multiple interactive real-time applications. On one hand, a CPU reservation mechanism is useful to ensure the QoS of soft real-time applications, and to provide temporal isolation for hard real-time applications. On the other hand, multicore technology provides additional computing power for heavy workloads, and the multicore scheduling policy determines how efficiently the system can use available CPU resources.

Unfortunately, there is not much study into the question of how well CPU schedulers in existing RTOSs can perform with multiple interactive real-time applications on multicore platforms. If we focus on those in Linux-based RTOSs, most prior work are evaluated by the experiments that run simple busy-loop tasks [11, 13, 15, 26], or only a few real-time applications on single-core platforms [27, 31]. The experimental results with only a few applications may not reflect the runtime performance under heavy workloads. Those with busy-loop tasks consuming the same amount of time in each pe-

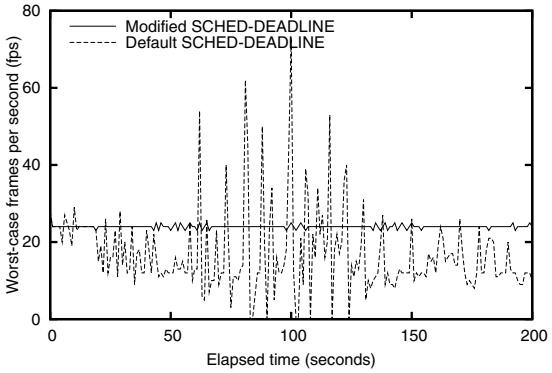


Figure 2. Frames per second achieved by the two versions of SCHED_DEADLINE, when playing the H264 (1920×800 at 24 fps) movie.

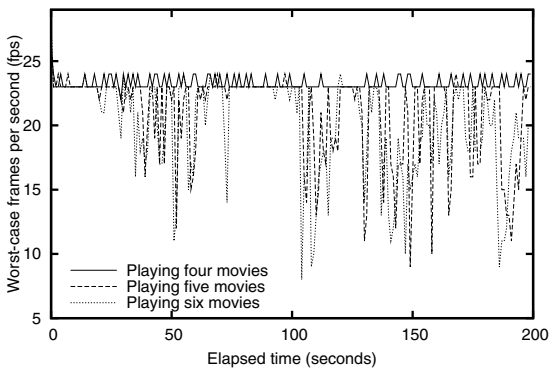


Figure 3. Worst-case frames per second achieved by SCHED_DEADLINE, when playing multiple H264 (1920×800 at 24 fps) movies.

riod may not also be applicable for real-world applications. For instance, Figure 1 shows the frame processing times of an H264-format movie with a frame size of 1920×800 and a frame rate of 24 frames per second (fps), achieved by an open-source movie player, mplayer, with the Linux kernel Version 2.6.32 on a 2.0 GHz Intel Core 2 Quad processor. It is observed that the frames require very different processing times. To assess the scheduler performance, we need to take into account this runtime behavior.

The question in this paper is how we provide adequate support for this kind of modern real-time applications. As mentioned above, there are practical CPU schedulers designed and implemented for real-time applications. In particular, SCHED_DEADLINE [18, 19] is a useful patch made for the Linux kernel, which implements an EDF scheduler with a CPU reservation mechanism based on Constant Bandwidth Server (CBS) [1, 2]. It is also available for multicore platforms. We therefore will use SCHED_DEADLINE, as a representative of the existing real-time schedulers.

First of all, we notice that the default reservation policy of SCHED_DEADLINE is slightly different from the original CBS algorithm. When we reserve CPU time Q (also called budget) per period P, its default policy guarantees CPU bandwidth strictly Q/P, while the original CBS algorithm does *at least* Q/P. In order to make it available for both hard and soft real-time applications, we modified its implementation to use the original CBS algorithm. Figure 2 shows the frames per second achieved by mplayer, using the modified and the default versions of SCHED_DEADLINE, when playing the H264 movie whose processing times are shown in Figure 1. We set $25ms$ to be the budget, since the worst-case value of around $45ms$ is too pessimistic to make efficient use of the available CPU bandwidth in this case, and it is rather appropriate to reserve the intermediate processing time per period. It is clear that the modified version of SCHED_DEADLINE is able to correctly ensure the QoS of the movie, whereas the default version repeatedly drops and catches up frames. However, even with the modified version of SCHED_DEADLINE, Figure 3 shows that the frame rates sometimes drop below 10 fps, when playing more than four instances of the H264 movie. Thus, the existing real-time scheduler is not adequate to maintain the QoS of the overall system, when running multiple interactive applications.

The contribution of this paper is to improve the capability of a CPU scheduler to maintain the QoS of the overall system, when multiple interactive real-time applications run on multicore platforms. To this end, we design and implement an **A**dvanced **I**nteractive and **R**eal-time **S**cheduler, called **AIRS**. As prior work [3, 18, 27, 31] do, AIRS incorporates the EDF algorithm and the CBS algorithm. It however provides new concepts: (i) a CPU reservation mechanism to improve the QoS of the overall system, when multiple applications reserve CPU bandwidth, and (ii) a multicore scheduler to improve the absolute CPU bandwidth available for the applications to perform well. We implement AIRS on top of SCHED_DEADLINE, given that (i) it is actively maintained now, (ii) it provides basic EDF scheduling features, and (iii) it is well-aligned with the mainline Linux kernel.

The remainder of this paper is organized as follows. In the next section, we discuss related work. In Section 3, the system model is defined. Section 4 presents a new CPU reservation mechanism, and Section 5 presents a new multicore scheduler, which are designed and implemented in AIRS. Section 6 evaluates the effectiveness of AIRS. We conclude this paper in Section 7.

2 Related Work

SCHED_DEADLINE [18, 19] is an implementation of the EDF algorithm for Linux, which is distributed as a kernel patch closely worked with the mainline Linux kernel. It also supports a CPU reservation mechanism and a multicore scheduler, which are respectively based on the classical CBS and Global EDF algorithms. AIRS is implemented on top of SCHED_DEADLINE to extend the capability of a CPU scheduler for interactive real-time applications.

LITMUS^{RT} [15] supports various multicore EDF-based schedulers. It has been used to discover new results on the implementation problems of multicore schedulers for real-time systems [11, 13]. It has also been used to evaluate adaptive QoS management [9] and resource synchronization [12]. Currently, its usage is limited to the Intel (x86-32) and Sparc64

architectures. In fact, it was reported [10] that the LITMUS^{RT} project was launched to bridge the gap between research results and industry practice. Meanwhile, the multicore scheduler supported by AIRS is designed and implemented to improve the CPU bandwidth available for applications with their timing requirements.

Redline [31] brings first-class support for interactive applications in commodity OSs. It offers memory and disk I/O management as well as CPU scheduling. The control group mechanism of the mainline Linux kernel and the EDF scheduler of SCHED_DEADLINE have nearly covered the Redline techniques. The primary objective of Redline is to ensure the interactivity when a set of applications is assigned sufficient CPU bandwidth in the system, while AIRS addresses the problem of maintaining the QoS of interactive real-time applications when they require CPU time more than they reserve under heavy workloads.

Linux-based RTOSs that exploit CPU reservation include Linux/RK [26], Linux-SRT [16], and AQuoSA [27]. Linux/RK implements both fixed-priority and EDF schedulers, though the EDF scheduler has not yet been extended to work on multicore platforms. It also supports a soft real-time CPU reservation policy. While this policy schedules the soft real-time tasks in background when their budgets are exhausted, AIRS assigns as early deadlines as possible to schedule these tasks by the EDF policy. Linux-SRT deals with disk I/O scheduling, like Redline, but is not designed to work on multicore platforms and to support an EDF scheduler. AQuoSA dynamically assigns CPU bandwidth to each task at runtime by a feedback controller, and therefore the scheduling performance highly depends on how a control model is suitable for a system. Meanwhile, AIRS assigns the constant CPU bandwidth to each task, and the tasks cooperate with each other by yielding their remaining budgets.

While the above prior work consider QoS management to ensure temporal isolation for applications, AIRS is aimed at improving the QoS of the overall system and the absolute CPU bandwidth available for applications to perform well.

RTLinux [32], RTAI [7], and KURT Linux [28] are particularly designed for hard real-time systems. Their requirements are to minimize the system latency. RED-Linux [29] provides a flexible framework for extensible scheduler implementation. It is focused on framework issues but not on performance issues. On the other hand, we consider QoS management and schedulability improvement.

3 Framework and Assumption

The system includes m CPUs: $\pi_1, \pi_2, ..., \pi_m$. Applications are composed of sporadic tasks. Each sporadic task τ_i is characterized by a period T_i, a relative deadline D_i, and a computation execution time C_i requested in each period. It generates a sequence of jobs. A job of τ_i released at time t is assigned a deadline at time $d_i = t + D_i$.

A server with a default budget Q_i and a period P_i is associated with each task τ_i to reserve CPU bandwidth Q_i/P_i. When the server is activated at time t, it is assigned a deadline at time $s_i = t + P_i$. A job of τ_i is scheduled based on the server deadline s_i. Note that it may be different from the real deadline d_i of τ_i. The remaining budget of τ_i is denoted by e_i. Like the prior work [18, 19, 27, 31], we associate one server with one task.

Table 1. APIs to control real-time applications.

`set_period`	sets the minimum inter-arrival time.
`set_deadline`	sets the relative deadline.
`set_runtime`	sets the CPU time reserved per period.
`wait_period`	suspends until the next period.

Each application uses a set of application programming interfaces (APIs) or system calls, which support at least the functions listed in Table 1. There may be additional APIs that provide other useful functions, such as synchronization, energy management, etc., though they are outside the scope of this paper. It is also not within the scope of this paper to discuss how they should be implemented. For example, some Linux-based RTOSs [7, 15, 18, 26, 31] implement these functions in the kernel, while others [27, 28] provide them in the Linux kernel module. In this paper, we implement the bulk of the functionality in the Linux kernel module for portability, and add the necessary change in the Linux kernel.

4 CPU Reservation Mechanism

Some interactive real-time applications are resource-intensive. Hence, CPU reservation is useful to ensure the QoS of those applications. As most prior EDF-based schedulers with CPU reservation mechanisms [3, 18, 19, 27, 31] do, AIRS incorporates the CBS-based algorithm to support QoS management. We however extend the CBS algorithm so that it can flexibly maintain the QoS of the overall system, when multiple real-time applications reserve CPU bandwidth. The idea behind the extension is to reclaim the remaining budget for other tasks to maintain the QoS of the overall system.

4.1 The CBS Algorithm

We briefly describe the reservation policy of the CBS algorithm below. Refer to [1, 2] for details.

- The server deadline and the budget are initialized by $s_i = 0$ and $e_i = 0$ respectively.
- When a job of a task τ_i is released at time t, if $e_i \geq (s_i - t)Q_i/P_i$, it is assigned a new server deadline at $s_i = t+P_i$, and the budget is replenished to $e_i = Q_i$.
- e_i is decreased by the same amount as the time consumed by the job of τ_i.
- When $e_i = 0$, it is replenished again to $e_i = Q_i$, and a new server deadline is set at $s_i = s_i + P_i$.

By the above reservation policy, it is guaranteed that a task τ_i receives CPU bandwidth of *at least* Q_i/P_i. However, the QoS management is restricted within a server. Even though the budget e_i remains when a job of τ_i completes, it is preserved until the next period for τ_i or it may be just discarded.

4.2 CPU Reservation Algorithm in AIRS

We propose a new CPU reservation algorithm, called *Flexible CBS* (FCBS). The FCBS algorithm uses a slack reclaiming approach, as also considered in the previous work [14, 22, 25]. Our approach is however distinguished from the previous work in that (i) it applies the same rule as the CBS algorithm for the budget replenishment and the deadline assignment, (ii) it guarantees each task τ_i to be assigned CPU bandwidth of at least Q_i/P_i even if the reclaiming occurs, and (iii) it reclaims the budgets so that the jobs lagging from the original EDF schedule are assigned more CPU bandwidth.

The FCBS algorithm donates the budgets left unused by some jobs to either the jobs lagging from the EDF schedule or the next earliest-deadline jobs. Let τ_i be a task whose job completes at time t. The remaining budget is e_i. According to the CBS algorithm, the CPU bandwidth used by τ_i does not exceed Q_i/P_i, even if the job of τ_i is executed for another e_i time units, with the current deadline s_i. Hence, the total CPU bandwidth used by all tasks do not also exceed $\sum_{\tau_k \in \tau} Q_k/P_k$, even though other jobs ready at time t with deadlines no earlier than s_i consume additional e_i time units. In addition, we never steal the budget from the future. As a result, it is obvious that the schedulability of the CBS algorithm is preserved.

The above observation leads to the following ideas. When some job of τ_i completes:

1. if there is a task τ_k whose server deadline is $s_k > t + P_k$, it means that τ_k has exhausted its budget at some earlier point of time, and is now lagging from the EDF schedule. In this case, we first save the original budget and the server deadline as $\tilde{e}_k = e_k$ and $\tilde{s}_k = s_k$ respectively. We then assign the budget $e_k = e_i$ and the deadline $s_k = s_i$ to τ_k temporarily, to catch up the EDF schedule. On one hand, if the job of τ_k can complete before exhausting e_k, the remaining budget e_k is succeeded to a different job again by the same policy. On the other hand, if τ_k exhausts e_k, the original budget and deadline are restored, i.e., $e_k = \tilde{e}_k$ and $s_k = \tilde{s}_k$. τ_k is then rescheduled.

2. if there is not such a task τ_k, we look for the next earliest-deadline task τ_j ready for execution, if any. We then add the remaining budget e_i to its budget e_j. In this case, we do not assign the server deadline s_i to τ_j, because it will be scheduled next. Even though some jobs are released with earlier deadlines before the job of τ_j completes, it can after all use the budget by its server deadline s_j.

3. otherwise, we preserve the remaining budget e_i as a bonus B. B is then decreased by the same amount as the idle time until a new job of some task τ_n is released. When the job of τ_n is released, we add the bonus B, if any remains, to the budget of τ_n according to the second policy described above.

The FCBS algorithm is easily extended for multicore scheduling. If tasks are globally scheduled so that at any time the m earliest deadline tasks, if any, are dispatched, the remaining budget can be used for any tasks in the system.

```
 1: function budget_exhausted_fcbs(τ_i) do
 2:   if s̃_i ≠ undefined then
 3:     e_i ← ẽ_i; s_i ← s̃_i; else e_i ← Q_i; s_i ← s_i + P_i;
 4:   end if
 5: end function
 6: function job_released_fcbs(τ_i) do
 7:   if B > 0 then
 8:     e_i ← e_i + B − min(t_now − t_last, B);
 9:     B ← 0;
10:   end if
11:   s̃_i ← undefined;
12: end function
13: function job_completed_fcbs(τ_i) do
14:   if s̃_i ≠ undefined then
15:     s_i ← s̃_i ; s̃_i ← undefined;
16:   end if
17:   j ← undefined;
18:   for each τ_k in order of early deadlines do
19:     if j = undefined then j ← k; end if
20:     if s_k > t_now + P_k then
21:       ẽ_k ← e_k; s̃_k ← s_k;
22:       e_k ← e_i; s_k ← s_i;
23:       return;
24:     end if
25:   end for
26:   if j ≠ undefined then
27:     e_j ← e_j + e_i; else B ← e_i; t_last ← t_now;
28:   end if
29: end function
```

Figure 4. Implementation of FCBS.

Meanwhile, if tasks are partitioned among CPUs, the remaining budget can only be used for the tasks on the same CPU.

Implementation. Figure 4 illustrates the pseudo-code of the implementation of the FCBS algorithm, added to SCHED_DEADLINE. In the pseudo-code, t_{now} denotes the current clock. The `budget_exhausted_fcbs` function is called when the budget of τ_i reaches zero. It is inserted at the end of the `deadline_runtime_exceeded` function in SCHED_DEADLINE. The `job_released_fcbs` function is called when a job of τ_i is released. It is inserted at the end of the `pick_next_task_deadline` function in SCHED_DEADLINE, but is executed only when τ_i holds the `DL_NEW` flag meaning that the task is starting a new period. The `job_completed_fcbs` function is called when a job of τ_i completes and suspends until the next period. It is inserted at the end of the `put_prev_task_deadline` function in SCHED_DEADLINE, but is also executed only when the task hold the `DL_NEW` flag.

5 Multicore Scheduler

Traditionally, there are two multicore scheduling concepts: *partitioned* and *global*. To simplify the description, we restrict our attention to the EDF algorithm. In partitioned scheduling, each task is assigned to a particular CPU, and is

scheduled on the local CPU according the EDF algorithm, without migration across CPUs. In global scheduling, on the other hand, tasks are scheduled so that the m earliest-deadline tasks in the system are executed on m CPUs, and hence they may migrate across CPUs. According to [18, 19], SCHED_DEADLINE leverages global scheduling.

Recently, several authors [5, 8, 21] have developed a new concept: *hybrid* of partitioned and global. Particularly, the Earliest Deadline First with Window-constraint Migration (EDF-WM) algorithm [21] is a simple design but still outperforms the existing EDF-based algorithms in terms of average schedulability, with a very small number of context switches.

In this section, we extend the EDF-WM algorithm so that it can work with the FCBS algorithm.

5.1 The EDF-WM Algorithm

We now briefly describe the EDF-WM algorithm. Refer to [21] for details. In the following, we assume that no tasks have reserves, for simplicity of description. That is, C_i, D_i and T_i are used to schedule each task τ_i.

When a new task τ_i is submitted to the system, the EDF-WM algorithm assigns a particular CPU to τ_i unless the total CPU bandwidth exceeds 100%, and τ_i never migrates across CPUs, like partitioned scheduling. If no such CPUs are found, it assigns multiple CPUs to τ_i. In this case, τ_i is allowed to migrate across those CPUs. Since an individual task cannot usually run on multiple CPUs simultaneously, the EDF-WM algorithm guarantees exclusive execution of τ_i on the multiple CPUs by splitting its relative deadline into the same number of windows as the CPUs. Let n_i be the number of the CPUs that are assigned to τ_i. Hence, the size of each window is equal to D_i/n_i. It allocates $C'_{i,x}$ as the CPU time allowed for τ_i to consume on CPU π_x every period T_i. According to [21], C'_i is given by

$$C'_{i,x} = \min\left\{\frac{L - \sum_{\tau_j \in \Gamma_x} \mathrm{dbf}(\tau_i, L)}{\lfloor (L - D_i/n_i)/T_i \rfloor + 1}\right\},$$

where Γ_x is a set of tasks assigned to CPU π_x, L is any value equal to the multiple of the relative deadline of each task $\tau_j \in \Gamma_x$, and $\mathrm{dbf}(\tau_j, L) = \max\{0, \lfloor (L - D_j/n_j)/T_j \rfloor + 1\} \times C'_j$.

At runtime, a job of τ_i released at time t on CPU π_x is assigned a pseudo-deadline $d'_i = t + D_i/n_i$ and is locally scheduled by the EDF policy until it consumes the CPU time $C'_{i,x}$. When $C'_{i,x}$ time units are consumed, τ_i is migrated to the next assigned CPU (let it be CPU π_y), and is again locally scheduled with a pseudo-deadline $d'_i = d'_i + D_i/n_i$ until it consumes $C'_{i,y}$ time units. If the sum of the assigned CPU time is less than C_i, τ_i is not guaranteed to be schedulable.

5.2 Multicore Scheduling Algorithm in AIRS

We now modify the EDF-WM algorithm for AIRS. The modified algorithm is called *EDF-WM and Reservation* (EDF-WMR). We assume that each task τ_i has a reserve managed by the FCBS algorithm. Thus, we replace C_i with Q_i, and both D_i and T_i with P_i, for each task τ_i. That is, if τ_i is a task that is allowed to migrate across multiple CPUs, the maximum CPU time $Q'_{i,x}$ allowed for τ_i to consume on CPU π_x at every period is given by

$$Q'_{i,x} = \min\left\{\frac{L - \sum_{\tau_j \in \Gamma_x} \mathrm{dbf}(\tau_i, L)}{\lfloor (L - P_i/n_i)/P_i \rfloor + 1}\right\},$$

where Γ_x is a set of tasks assigned to CPU π_x, L is any value equal to the relative deadline of any task $\tau_j \in \Gamma_x$, and $\mathrm{dbf}(\tau_j, L) = \max\{0, \lfloor (L - P_j/n_j)/P_j \rfloor + 1\} \times C'_j$. Note that we need not search for those values of L that are the multiples of the relative deadline of each task, if relative deadlines are equal to periods, according to [21]. Since both D_i and T_i are replaced with P_i in the FCBS algorithm, we can have the above definitions.

Reservation for migratory tasks. We first examine how the original EDF-WM algorithm can work with the FCBS algorithm. Let τ_i be a task allowed to migrate across multiple CPUs. We consider the case in which τ_i is activated on CPU π_x at time t, either when it is migrated to or is released on the CPU. The budget is set to $e_i = Q'_{i,x}$. That is, τ_i is allowed to consume $Q'_{i,x}$ time units on π_x, with a server deadline $s_i = t + P_i/n_i$. Being subject to the original EDF-WM algorithm, when $Q'_{i,x}$ is exhausted, τ_i is migrated to another CPU π_y. The budget is then replenished to $e_i = Q'_{i,y}$, and the server deadline is set to $s_i = s_i + P_i/n_i$.

We next modify the above reservation policy to improve the runtime performance, when multiple tasks are scheduled with reserves. When $Q'_{i,x}$ is exhausted on CPU π_x, the EDF-WMR algorithm does not migrate τ_i to another CPU π_y but strictly postpones the migration until $t + P_i/n_i$. It then sets the server deadline to $s_i = s_i + P_i = t + P_i/n_i + P_i$, and holds the budget $e_i = 0$. Since the budget is not replenished by $Q'_{i,x}$, it does not affect the reservation in the next period. Instead, since $s_i > t + P_i$ is satisfied at any time $t < t + P_i/n_i$, the budget may be replenished by the remaining budget of another completed job, according to the FCBS algorithm. At time $t' = t + P_i/n_i$, the server deadline is reset to $s_i = t'$, and τ_i is migrated to another CPU π_y. The server deadline is then set to $s_i = s_i + P_i/n_i$ for the execution on π_y. Note that the schedulability is independently guaranteed on each CPU by the EDF-WM algorithm. Hence, the schedulability is not affected on π_y, even if we reset the server deadline to $s_i = t'$ on π_x before the migration. That is, τ_i is guaranteed to be schedulable on π_y by the EDF-WMR algorithm, with the arrival time at t' and the deadline $s_i = t' + P_i/n_i$.

Heuristics for CPU assignment. The original EDF-WM algorithm uses the first-fit (FF) heuristic that assigns the first CPU to a new task, upon which the schedulability test is passed. In fact, most existing EDF algorithms [6, 8, 24] based on partitioned scheduling use the FF heuristic, since it bounds the worst-case schedulability [24]. However, in AIRS, we use the worst-fit (WF) heuristic that assigns the CPU with the lowest utilization to a new task, in order to make the scheduler more robust for reservation-based systems as well as miss specifications of task parameters.

The WF heuristic performs load-balancing across all CPUs, while the FF heuristic tries to pack earlier CPUs fully,

```
1:  function budget_exhausted_wmr(τ_i) do
2:      if τ_i is a migratory task then e_i ← 0; end if
3:  end function
4:  function job_released_wmr(τ_i) do
5:      if τ_i is a migratory task then
6:          e_i ← Q'_{i,x};
7:          start_timer(migration(τ_i), t_now + P_i/n_i);
8:      end if
9:  end function
10: function job_completed_wmr(τ_i) do
11:     if τ_i is a migratory task then
12:         migrate τ_i to the first CPU assigned to τ_i;
13:     end if
14: end function
15: function migration(τ_i) do
16:     s_i ← t_now + P_i/n_i;
17:     migrate τ_i to the next CPU assigned to τ_i;
18:     start_timer(migration(τ_i), t_now + P_i/n_i);
19: end function
20: function start_timer(func(), t) do
21:     run a timer to invoke func() at time t;
22: end function
```

Figure 5. Implementation of EDF-WMR.

leaving later CPUs less packed. As a result, under the WF heuristic, the utilization of each CPU is close to the average utilization, and average response times tend to be better.

Consider a problem in which only one CPU π_1 executes real-time tasks and its CPU utilization is U_1. Then, a new task τ_i is submitted to the system with $Q_i/P_i = 1 - U_x$. If we use the FF heuristic, CPU π_1 is assigned to τ_i. However, Q_i may be much smaller than the actual execution time in a reservation-based system, or with a miss specification. As a result, CPU π_x can be transiently overloaded at runtime, and the tasks running on π_x may miss deadlines, despite the remaining CPU resources on $m-1$ CPUs. On the other hand, the WF heuristic does not assign CPU π_1 but another CPU to τ_i. Hence, unexpected deadline misses are avoided. Even with the WF heuristic, the schedulability is still guaranteed with Q_i and P_i, since the EDF-WM algorithm is known to be effective for any CPU assignment heuristics. In Section 6, we show that the WF heuristic performs well for AIRS.

Implementation. Figure 5 illustrates the pseudo-code of the implementation of the EDF-WMR algorithm, added to SCHED_DEADLINE with FCBS. Since the pseudo-code to assign CPUs to a migratory task was already shown in [21], we only focus on the runtime scheduling functions. In the pseudo-code, we assume that τ_i is released on π_x. The `budget_exhausted_wmr`, `job_released_wmr`, and `job_completed_wmr` functions are appended at the ends of the `budget_exhausted_fcbs`, `job_released_fcbs`, and `job_completed_fcbs` functions respectively, which are shown in Figure 4. The `job_migration` function internally uses the `set_cpus_allowed_ptr` function, provided by the Linux kernel, to migrate tasks. The `start_timer` function also uses the timer functions provided by the Linux kernel. Specifically, high-resolution timers are used if enabled. Else, the default tick-driven timers are used.

6 Experimental Evaluation

We now present a quantitative evaluation of AIRS. All measurements are performed on a system with a 2.0 GHz Intel Core 2 Quad processor (Q9650), 2 GB of DDR2-SDRAM, and an Intel GMA 4500 integrated graphics card. The Linux kernel Version 2.6.32 patched with SCHED_DEADLINE is used as the underlying OS. To the best of our knowledge, no scheduler distributions other than SCHED_DEADLINE implement EDF schedulers with CPU reservation mechanisms, which are designed to work on the above system. Hence, we compare AIRS with SCHED_DEADLINE, as a representative of the existing EDF-based schedulers. As we implied in Section 1, SCHED_DEADLINE is modified to use the original CBS algorithm [1, 2].

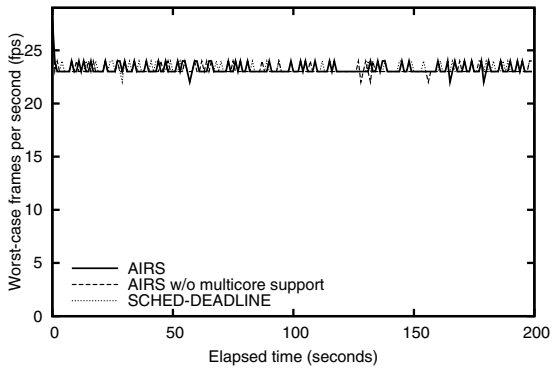
Experiments with movie player. The open-source movie player, mplayer, is used to evaluate the ability of AIRS with respect to the achievable QoS of applications. In our experiments, mplayer is modified from the original source so that it uses the APIs listed in Table 1. The modification is, however, limited to only five lines of the source code and one additional header file. We always run mplayer as a real-time task in Linux so as not to be affected by background activities.

The experiments play two types of movies. One is compressed by the H264 standard, with a frame size of 1920×800 and a frame rate of 24 fps (a period is $41ms$). Playing this type of movie evaluates the case in which there are big differences among the average-case, the best-case, and the worst-case execution times. The other is an uncompressed movie, with a frame size of 720×480 and a frame rate of 29 fps (a period is $33ms$). Playing this type of movie evaluates the case in which the execution times of jobs do not vary very much.

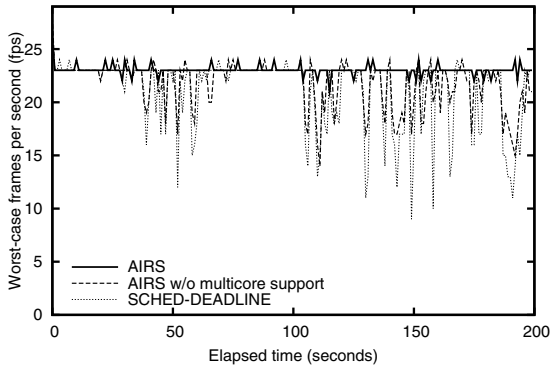
We first conduct experiments on playing multiple instances of the H264 movie. Its frame processing times are shown in Figure 1. It is observed that a majority of frames are processed within $25ms$. To ensure the QoS of this majority, we set $25ms$ to be the CPU time reserved per period for both AIRS and SCHED_DEADLINE.

To see how the FCBS and the EDF-WMR algorithms improve the runtime performance respectively, we run two versions of AIRS. One uses only the FCBS algorithm, and the other uses both of the algorithms. When AIRS runs without multicore support, it relies on SCHED_DEADLINE to dispatch tasks to CPUs. Since our goal is to ensure the QoS, we do not use the EDF-WM algorithm alone.

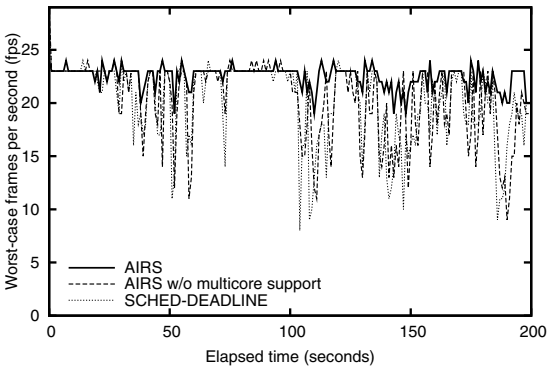
Figure 6 shows the worst number of frames per second, when playing multiple instances of the H264 movie. Note that we pick the worst number of frames in all the played movies at every sampling point, so the results are not focused on a particular movie but indicate the QoS of the overall system. Figure 6(a) depicts the results obtained by playing four movies. In this case, both AIRS and SCHED_DEADLINE are capable of achieving high frame rates, regardless of mul-



(a) Playing four movies



(b) Playing five movies



(c) Playing six movies

Figure 6. Worst-case frames per second when playing multiple instances of the H264 movie.

ticore support. Since we have four CPUs (a quad-core machine), one CPU is exclusively assigned to each mplayer instance, even though the tasks are globally scheduled by SCHED_DEADLINE. Thus, they show very competitive performance. The frame rates occasionally drop below 24 fps. This is because some individual frames require the processing time more than the period, as shown in Figure 1. The frame processing times are affected by the cache status, and it is not predictable due to the existence of Linux background jobs. Hence, AIRS may have more frames per second than SCHED_DEADLINE, and vice versa.

Figure 6(b) shows the worst-case frames per second when playing five H264 movies. In this case, the results

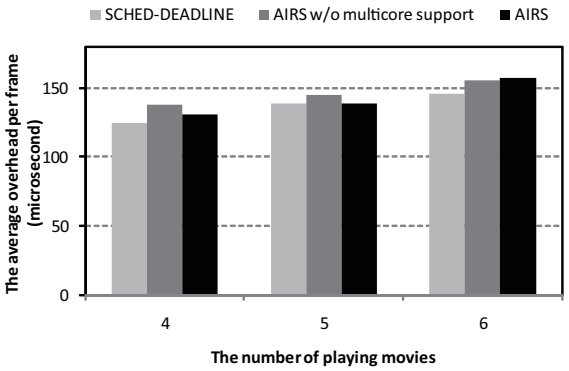


Figure 7. Average system overheads for a single frame of the H264 movie.

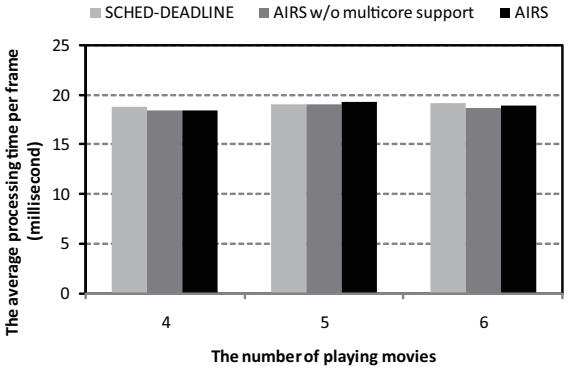


Figure 8. Average frame processing times of the H264 movie.

demonstrate that AIRS outperforms SCHED_DEADLINE. While AIRS with multicore support is still able to maintain nearly 24 fps, AIRS without multicore support and SCHED_DEADLINE decrease the frame rates. It is also worth noting that AIRS without multicore support is better than SCHED_DEADLINE. This means that the FCBS algorithm is more efficient than the CBS algorithm in maintaining the QoS of the overall system. The multicore support in AIRS brings further performance improvements. Given that there are five instances of mplayer, each of which reserves CPU bandwidth $25/41 = 61\%$, they need to migrate across CPUs to meet deadlines. In this scenario, the timely migration mechanism provided by the EDF-WMR scheduler in AIRS correctly schedules the corresponding tasks, while the global EDF scheduler in SCHED_DEADLINE causes the tasks to miss deadlines more frequently. Hence, we conclude that a hybrid of partitioned and global multicore scheduling approaches is also effective for QoS-aware real-time applications, beyond hard timing guarantees assessed in [21].

Figure 6(c) shows the worst-case frames per second when playing six H264 movies. The results show that AIRS without multicore support decreases the frame rates as much as SCHED_DEADLINE. This means that the global EDF scheduler in SCHED_DEADLINE is no more capable of scheduling the tasks, even though the FCBS algorithm is applied. On the other hand, AIRS with multicore support still delivers high quality to all the six movies. In fact, the six tasks

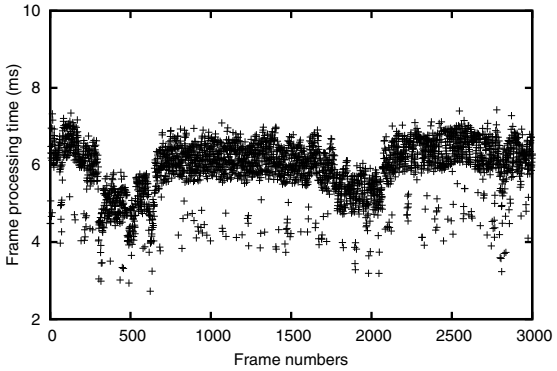


Figure 9. Frame processing times of an uncompressed (720×480 at 29 fps) movie.

with computation time $25ms$ and period $41ms$ are theoretically schedulable by the EDF-WM algorithm. Even if their jobs spend more than $25ms$ of CPU time, the FCBS algorithm assists the overrun. As a result, the tasks remains without fatal timing violation.

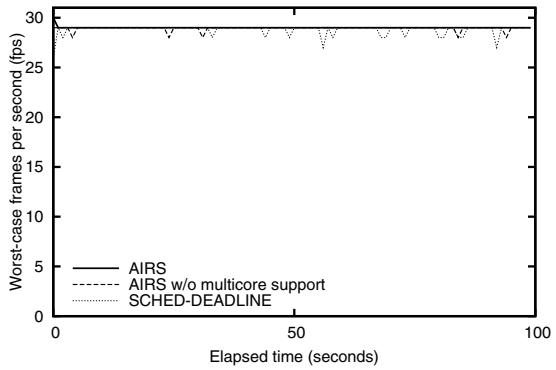
To our experience, when the frame rates drop below 20 fps, we visually notice that the movies are delayed. When the frame rates drop below 15 fps, we start feeling uncomfortable watching the movies. In this sense, AIRS is useful to provide sufficient quality to interactive real-time applications.

Figure 7 shows the average system overheads taken to process a single frame of the H264 movie. It is measured by the *stime* member variable in the task descriptor of the Linux kernel. We observe that the additional implementation overhead of AIRS is negligibly small as compared to SCHED_DEADLINE. The maximum difference is no more than 15 microseconds. The absolute system overheads increase as the numbers of playing movies increase, since the scheduling decision clearly spends more CPU time to deal with more tasks. It is also interesting that the implementation of the EDF-WMR algorithm reduces the system overheads for some cases. We consider that each local scheduler only deals with the tasks assigned to the corresponding CPU by partitioning the tasks among CPUs, and hence the scheduling decision needs less CPU time than global scheduling.

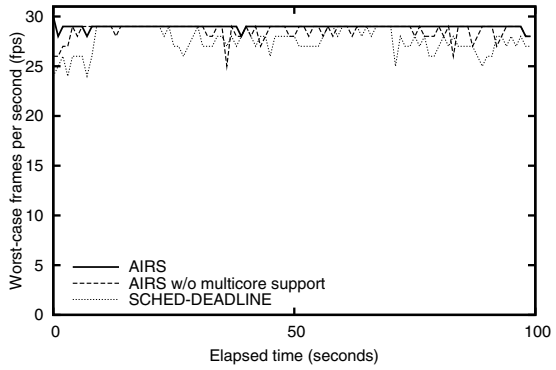
Figure 8 shows the average frame processing times of the H264 movie. We observe that the average frame processing times are almost the same in all cases. This means that the memory cache affects all cases equally. Although we have not traced the cache behavior precisely, we believe that the numbers of context switches and migrations are not so different in those measurements as to change the cache performance, since we run only four to six tasks.

We next evaluate the effectiveness of AIRS in playing multiple instances of the uncompressed movie. Its frame processing times are shown in Figure 9. Since the frames are not compressed, the processing times do not fluctuate as much as the H264 movie. Based on the obtained processing times, we set $9ms$ to be the CPU time reserved per period, which is enough to ensure the QoS of the overall system.

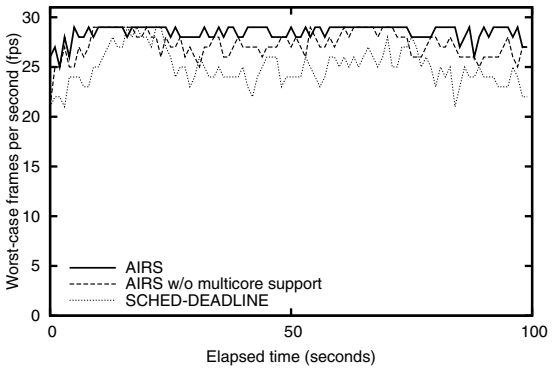
Figure 10 shows the worst number of frames processed per second, when playing multiple instances of the uncompressed



(a) Playing twelve movies



(b) Playing thirteen movies



(c) Playing fourteen movies

Figure 10. Worst-case frames per second when playing multiple instances of the uncompressed movie.

movie. Since the frame processing times are far smaller than those of the H264 movie, the system can play more movies. Given that the CPU bandwidth required by each mplayer instance is $9/33 \simeq 27\%$, the system should have enough CPU bandwidth when playing mo more than fourteen movies.We observe that the achieved frame rates are not very different between AIRS and SCHED_DEADLINE in all cases. This means that the CBS algorithm is also as effective in ensuring the QoS of the overall system as the FCBS algorithm, when applications request CPU time less than they reserve. However, we still obtain the performance differences between them, when playing more than twelve movies. It is clear

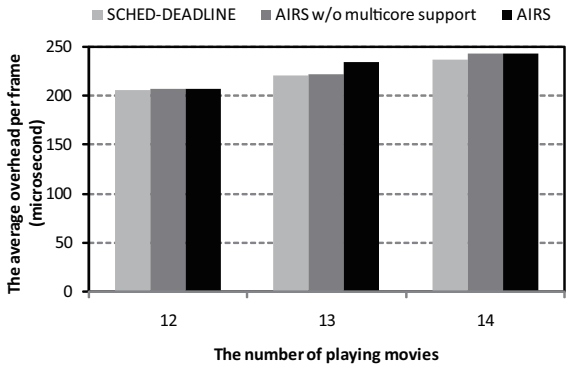


Figure 11. Average system overheads for a single frame of the uncompressed movie.

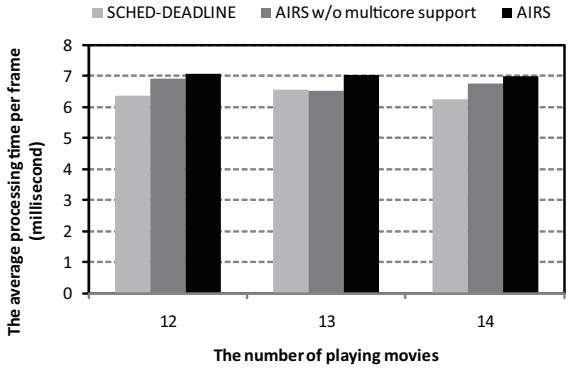


Figure 12. Average frame processing times of the uncompressed movie.

that these performance differences arise from their respective multicore schedulers. Hence, the results demonstrate that the timely migration mechanism based on the EDF-WMR algorithm is effective to ensure the QoS of the overall system, even when the frame processing times do not change very much.

Figure 11 shows the average system overloads taken to process a single frame of the uncompressed movie. The absolute system overheads are increased in all cases, as compared to the case for the H264 movie shown in Figure 7. The increase in the system overheads mostly include the scheduling overheads, because there are more than twelve tasks to schedule. However, we claim that the difference between AIRS and SCHED_DEADLINE is still ignorable.

Figure 12 shows the average frame processing times of the uncompressed movie. Surprisingly, AIRS needs the most CPU time to process a single frame, despite its limited migrations. Even if the multicore support is not activated, it needs more processing time than SCHED_DEADLINE. It is reasoned as follows. First, the FCBS algorithm may suspend and resume the same job repeatedly. When it receives the remaining budget from other completed jobs, it is resumed. When the budget is exhausted, it is suspended. Hence, we consider that the memory cache affects the processing times due to a number of context switches, unlike the previous cases scheduling no more than six tasks. In particular, AIRS with multicore support assigns particular CPUs to the tasks, so the FCBS algorithm is applied independently on each CPU. As a result, the cache performance is affected more. However, AIRS improves the frame rates after all.

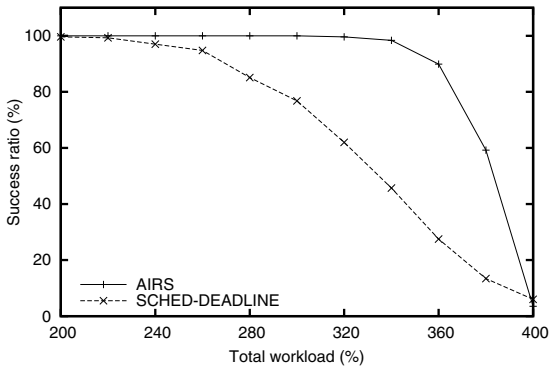


Figure 13. Success ratios of scheduling busy-loop tasks with respect to workloads.

Experiments with busy-loop tasks. We now assess the breakdown workload for hard deadline guarantees, by scheduling large sets of randomly-generated busy-loop periodic tasks, like prior work [11, 13, 15]. The execution times are tightly bounded in this case. Thus, we do not use CPU reservation mechanisms for the following experiments.

We submit 1000 sets of periodic tasks, each of which produces the same amount of workload W, to measure schedulability for the given workload. Each task set is generated as follows. The CPU utilization U_i of a new task τ_i is determined based on a uniform distribution. Due to space constraints, we only show the results of the tests conducted with the range $[10\%, 100\%]$. New tasks are created until the total CPU utilization reaches W. The period T_i of τ_i is also uniformly determined within the range of $[1000, 100000]$, assuming that applications have periods ranging from $1ms$ to $100ms$. The deadline is set equal to the period, and the execution time is computed as $C_i = U_i T_i$.

When the task parameters are decided, we measure the count n of busy-loops that consume 1 microsecond. Each task τ_i then loops $n \times C_i$ times in each period. We run the tasks for 10 minutes. A task set is said to be successfully scheduled if and only if no tasks miss deadlines during the measurement. The hard real-time performance is evaluated as the success ratio: *the ratio of the number of successfully-scheduled task sets and that of task sets tested for schedulability*.

Figure 13 shows the success ratios of scheduling the task sets with the given workloads. We note again that each sampling point runs 1000 task sets. According to the results, AIRS also has an advantage in guaranteeing hard deadlines. It is known [17] that the global EDF algorithm shows worse schedulability when high-utilization tasks exist. Since the experiment includes them, the results reflect this effect.

7 Conclusions

We have presented AIRS, an Advanced Interactive and Real-time Scheduler, for supporting interactive real-time applications on multicore platforms. It incorporates a new CPU reservation algorithm, called FCBS, to improve the QoS of

the overall system, when multiple applications reserve CPU bandwidth. It also employs a new multicore scheduling algorithm, called EDF-WMR, to improve the absolute CPU bandwidth available for applications to perform well. They have been implemented on top of SCHED_DEADLINE [18, 19].

Our detailed experimental evaluation showed that AIRS is able to achieve higher frame rates than SCHED_DEADLINE, when running multiple movies with heavy workloads on multicore platforms, and the additional implementation overhead of AIRS is negligibly small. It also showed that AIRS is able to improve the breakdown workload for hard timing guarantees, when running randomly-generated task sets, as compared to SCHED_DEADLINE.

AIRS is open-source software, and may be downloaded from http://www.ece.cmu.edu/~shinpei/airs/. The performance evaluation tools used in the experiments may also be downloaded from the website. They are currently available for AIRS, SCHED_DEADLINE, Linux/RK, and LITMUS^{RT}.

References

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proc. of the IEEE Real-Time Systems Symposium*, pp. 3–13, 1998.

[2] L. Abeni and G. Buttazzo. Resource reservation in dynamic real-time systems. *Real-Time Systems*, pp. 123–167, 2004.

[3] L. Abeni and G. Lipari. Implementing resource reservations in Linux. In *Real Time Linux Workshop*, 2002.

[4] K. Akachi, K. Kaneko, N. Kanehira, S. Ota, G. Miyanori, M. Hirata, S. Kajita, and F. Kanehiro. Development of humanoid robot HRP-3P. In *Proc. of the IEEE-RAS International Conference on Humanoid Robots*, pp. 50–55, 2005.

[5] B. Andersson, K. Bletsas, and S. Baruah. Scheduling arbitrary-deadline sporadic task systems on multiprocessors. In *Proc. of the IEEE Real-Time Systems Symposium*, pp. 385–394, 2008.

[6] T. Baker. A comparison of global and partitioned EDF schedulability tests for multiprocessors. In *Proc. of the International Conference on Real-Time and Network Systems*, pp. 119–127, 2006.

[7] D. Beal, E. Bianchi, L. Dozio, S. Hughes, P. Mantegazza, and S. Papacharalambous. RTAI: Real Time Application Interface. *Linux Journal*, 29:10, 2000.

[8] K. Bletsas and B. Andersson. Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. In *Proc. of the IEEE Real-Time Systems Symposium*, pp. 447–456, 2009.

[9] A. Block, B. Brandenburg, J. Anderson, and S. Quint. Adaptive multiprocessor real-time scheduling with feedback control. In *Proc. of the Euromicro Conference on Real-Time Systems*, pp. 23–33, 2008.

[10] B. Brandenburg and J. Anderson. Joint opportunities of real-time Linux and real-time systems research. In *Proc. of the Real-Time Linux Workshop*, 2009.

[11] B. Brandenburg and J. Anderson. On the implementation of global real-time schedulers. In *Proc. of the IEEE Real-Time Systems Symposium*, pp. 214–224, 2009.

[12] B. Brandenburg and J. Anderson. Reader-writer synchronization for shared-memory multiprocessor real-time systems. In *Proc. of the Euromicro Conference on Real-Time Systems*, pp. 184–193, 2009.

[13] B. Brandenburg, J. Calandrino, and J. Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Proc. of the IEEE Real-Time Systems Symposium*, pp. 157–169, 2008.

[14] M. Caccamo, G. Buttazzo, and D. Thomas. Efficient reclaiming in reservation-based real-time systems. *Real-Time Systems*, 54(2):198–213, 2005.

[15] J. Calandrino, H. Leontyev, A. Block, U. Devi, and J. Anderson. LITMUS^{RT}: A testbed for empirically comparing real-time multiprocessor schedulers. In *Proc. of the IEEE Real-Time Systems Symposium*, pp. 111–123, 2006.

[16] S. Childs and D. Ingram. The Linux-SRT integrated multimedia operating systems: Bringing QoS to the desktop. In *Proc. of the IEEE Real-Time Technology and Applications Symposium*, pp. 135–140, 2001.

[17] S. K. Dhall and C. L. Liu. On a real-time scheduling problem. *Operations Research*, 26:127–140, 1978.

[18] D. Faggioli, M. Trimarchi, and F. Checconi. An implementation of the Earliest Deadline First algorithm in Linux. In *Proc. of the ACM symposium on Applied Computing*, pp. 1984–1989, 2009.

[19] D. Faggioli, M. Trimarchi, F. Checconi, and C. Scordino. An EDF scheduling class for the Linux kernel. In *Proc. of the Real-Time Linux Workshop*, 2009.

[20] K. Kaneko, F. Kanehiro, H. Hirukawa, T. Kawasaki, M. Hirata, K. Akachi, and T. Isozumi. Humanoid robot HRP-2. In *Proc. of the IEEE International Conference on Robotics and Automation*, pp. 1083–1090, 2004.

[21] S. Kato, N. Yamasaki, and Y. Ishikawa. Semi-partitioned scheduling of sporadic task systems on multiprocessors. In *Proc. of the Euromicro Conference on Real-Time Systems*, pp. 249–258, 2009.

[22] C. Lin and S. Brandt. Improving soft real-time performance through better slack reclaiming. In *Proc. of the IEEE Real-Time Systems Symposium*, pp. 410–421, 2005.

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

[24] J. Lopez, J. Diaz, and D. Garcia. Utilization bounds for EDF scheduling on real-time multiprocessor systems. *Real-Time Systems*, 28:39–69, 2004.

[25] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo. IRIS: A new reclaiming algorithm for server-based real-time systems. In *Proc. of the IEEE Real-Time and Embedded Technology and Aplications Symposium*, pp. 211–218, 2004.

[26] S. Oikawa and R. Rajkumar. Portable RT: A portable resource kernel for guaranteed and enforced timing behavior. In *Proc. of the IEEE Real-Time Technology and Aplications Symposium*, pp. 111–120, 1999.

[27] L. Palopoli, T. Cucinotta, L. Marzario, and G. Lipari. AQuoSA: Adaptive quality of service architecture. *Software–Practice and Experience*, 39:1–31, 2009.

[28] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus. A firm real-time system implementation using commercial off-the-shelf hardware and free software. In *Proc. of the IEEE Real-Time Technology and Applications Symposium*, pp. 112–119, 1998.

[29] Y. Wang and K. Lin. Implementing a general real-time scheduling framework in the RED-Linux real-time kernel. In *Proc. of the IEEE Real-Time Systems Symposium*, pp. 246–255, 1999.

[30] M. Yamashita, R. Sakai, A. Tanaka, K. Imada, Y. Takahashi, T. Ida, N. Matsumoto, and N. Kato. AV applications for TV sets empowered by Cell Broadband Engine. In *Proc. of the International Conference on Consumer Electronics*, pp. 1–2, 2009.

[31] T. Yang, T. Liu, E. Berger, S. Kaplan, and J.-B. Moss. Redline: First class support for interactivity in commodity operating systems. In *Proc. of the USENIX Symposium on Operating Systems Design and Implementation*, pp. 73–86, 2008.

[32] V. Yodaiken. The rtlinux manifesto. In *Proc. of the Linux Expo*, 1999.