Noname manuscript No.
(will be inserted by the editor)

CPU Scheduling and Memory Management for Interactive Real-Time Applications

Shinpei Kato · Yutaka Ishikawa · Ragunathan (Raj) Rajkumar

Received: date / Accepted: date

Abstract In this paper, we propose, design, implement, and evaluate a CPU scheduler and a memory management scheme for interactive soft real-time applications. Our CPU scheduler provides a new CPU reservation algorithm that is based on the well-known Constant Bandwidth Server (CBS) algorithm but is more flexible in allocating the CPU time to multiple concurrently-executing real-time applications. Our CPU scheduler also employs a new multicore scheduling algorithm, extending the Earliest Deadline First to yield Window-constraint Migrations (EDF-WM) algorithm, to improve the absolute CPU bandwidth available in reservation-based systems. Furthermore, we propose a memory reservation mechanism incorporating a new paging algorithm, called Private-Shared-Anonymous Paging (PSAP). This PSAP algorithm allows interactive real-time applications to be responsive under memory pressure without wasting and compromising the memory resource available for contending best-effort applications. Our evaluation demonstrates that our CPU scheduler enables the simultaneous playback of multiple movies to perform at the stable frame-rates more than existing real-time CPU schedulers, while also improves the ratio of hard-deadline guarantee for randomly-generated task sets. Furthermore, we show that our memory management scheme can protect the simultaneous playback of multiple movies from the interference introduced by memory pressure, whereas these movies can become unresponsive under existing memory management schemes.

S. Kato and R. Rajkumar
Department of Electrical and Computer Engineering, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213 USA
E-mail: {shinpei,raj}@ece.cmu.edu

Y. Ishikawa
Department of Computer Science, The University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8654 JAPAN
E-mail: ishikawa@is.s.u-tokyo.ac.jp

1 Introduction

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

Modern real-time systems have become increasingly resource-intensive and interactive. For instance, the state-of-the-art humanoid robots execute both hard and soft real-time applications concurrently (Akachi et al, 2005; Kaneko et al, 2004), and digital high-definition TVs are also expected to display multiple video contents at the same time (Yamashita et al, 2009). Other examples include teleconferencing, video streaming, and 3-D online games. The operating system (OS) for these systems needs to provide resource management schemes that can enhance and maintain the QoS of the overall system even under heavy workloads.

CPU scheduling is a central concept in real-time OSes (RTOSes) to meet timing requirements. In particular, CPU reservation and multicore scheduling are key techniques to further support concurrently-executing multiple interactive real-time applications. CPU reservation is useful to maintain the QoS for soft real-time applications, and it is also beneficial to provide temporal isolation for hard real-time applications. Multicore technology, meanwhile, supplies additional computing power for heavy workloads, and multicore scheduling algorithms determine how efficiently the system can utilize CPU resources.

Unfortunately, there is not much study into the question of how well CPU schedulers in existing RTOSes can perform with concurrently-executing multiple interactive real-time applications on multicore platforms. Consider those in Linux-based RTOSes. Most prior approaches are evaluated by experiments or simulations that simply execute busy-loop tasks (Brandenburg and Anderson, 2009b; Brandenburg et al, 2008; Calandrino et al, 2006; Oikawa and Rajkumar, 1999), or just a few real-time applications on single-core platforms (Palopoli et al, 2009; Yang et al, 2008). Experimental results obtained from executing only a few applications may not demonstrate CPU scheduler capabilities under heavy workloads. On the other hand, those obtained from executing busy-loop tasks that consume the same amount of time in each period may not be applicable for real-world scenarios. As a consequence, the scheduling problem for concurrently-executing multiple interactive real-time applications is still left open for multi-core platforms.

The first question we try to address in this paper is how to provide adequate support for scheduling of interactive real-time applications on multi-core platforms. SCHED_DEADLINE (Faggioli et al, 2009a,b) is a real-time CPU scheduler developed for the Linux kernel, which implements (i) Earliest Deadline First (EDF) (Liu and Layland, 1973) as a scheduling algorithm and (ii) Constant Bandwidth Server (CBS) (Abeni and Buttazzo, 1998, 2004) as a CPU reservation algorithm. It is also available for multicore platforms. Our preliminary evaluation on a 2.0 GHz Intel Core 2 Quad processor (Q9650) has demonstrated that the frame-rate of H.264 movies

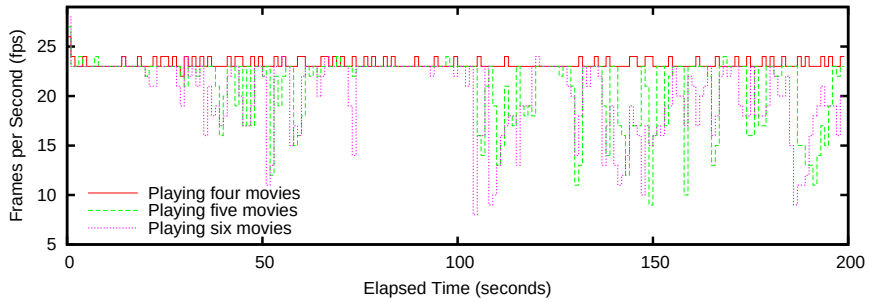

Fig. 1 Frame-rates of movies playing concurrently under SCHED_DEADLINE on a quad-core CPU.

played by the MPlayer open-source movie player could drop below 10 fps under SCHED_DEADLINE, when playing more than four instances of the movie, even though the system is not over-utilized, as shown in Figure 1, where the average-case execution time is used as a reservation capacity for the CBS algorithm. This is largely attributed to the fact that the MPlayer task has a variable execution time for each frame, and hence constant-bandwidth reservation is not able to handle well such frames that try to run out of reservation. The EDF algorithm could also perform poorly on multi-core platforms (Dhall and Liu, 1978). It is not hard to see from this preliminary evaluation that existing real-time CPU schedulers are inadequate when executing multiple interactive applications concurrently on multi-core platforms. To the best of our knowledge, no other scheduler distributions employ both the EDF and the CBS algorithms available for multi-core platforms. We will therefore use SCHED_DEADLINE as a representative of existing real-time CPU schedulers.

Memory management is another concern for interactive real-time applications that tend to be memory-intensive. Even though CPU schedulers provide adequate support, the system can still easily become unresponsive by memory-intensive activities. Figure 2 shows the number of swapped-out pages of the MPlayer task, when contending tasks are allocating 2GB of memory resource in total. Some frames have no less than about 400 pages to be forcefully swapped out, whereas no page swapping has been observed when the MPlayer task executes alone. Given that a disk access time to swap out one page could be around a few milliseconds, the traditional least-recently-used (LRU) paging algorithm imposes unacceptable performance penalties to interactive real-time applications.

The second question we try to address in this paper is therefore “What is an appropriate memory management scheme for interactive real-time applications”? Previous real-time systems conservatively use a memory locking (e.g., *mlock*) mechanism that pins all the pages used by critical applications (Laplante, 1993). Unfortunately, this approach is not very effective in the above scenario, since some pages still need to be swapped out when locking new pages, if there are no free pages due to memory pressure. Memory reservation strategies, such as memory firewalling (Eswaran and Rajkumar, 2005; Hand, 1999) and the Linux *memory cgroup*, provide better responsiveness for interactive real-time applications. Unlike memory locking, memory reservation ensures that the memory blocks reserved by some applications are never

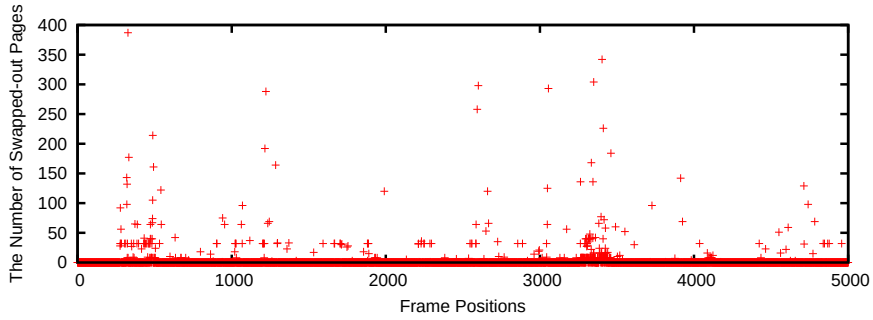

Fig. 2 The number of swapped-out pages in playing the H.264 movie under memory pressure.

accessed by others. However, previous memory reservation mechanisms still have the downside that applications become unresponsive quickly, if all pages in their reserved memory blocks are in use, meaning that they need to swap out some pages.

The primary contribution of this paper is improving the capabilities of a CPU scheduler and a memory management scheme for interactive soft real-time applications. We particularly focus on maintaining concurrently-executing multiple soft real-time applications at the desired frame-rates even in the face of heavy workloads and contending background resource-intensive activities. While our CPU scheduler is based on the well-known EDF and the CBS algorithms, this paper provides new concepts: (i) a CPU reservation algorithm that improves the QoS of the overall system when multiple applications try to reserve CPU bandwidth concurrently, and (ii) a multicore scheduling algorithms that improves the absolute CPU bandwidth available for these multiple applications to reserve CPU bandwidth. We implement our CPU scheduler using SCHED_DEADLINE, given that (i) it has been actively maintained, (ii) it provides basic EDF scheduling features, and (iii) it is well-aligned with the mainline of the Linux kernel development. We also design and implement a memory reservation mechanism that makes soft interactive real-time applications responsive, even if the pages in memory blocks reserved by these applications are exhausted.

The rest of this paper is organized as follows. Section 2 discusses the related work in this area. Section 3 describes our system model. Section 4 and Section 5 present the CPU reservation and the multicore scheduling algorithms used in our CPU scheduler respectively. Section 6 describes the reservation mechanism in our memory management scheme. The capabilities of our CPU scheduler and memory management scheme are evaluated in Section 7. Section 2 discusses the related work in this area. We provide our concluding remarks in Section 8.

2 Related Work

In this section, we briefly discuss related work with respect to the areas of Linux-based RTOSes, CPU scheduling, and memory management for interactive real-time applications. For CPU scheduling, however, we only focus on those designed based on the EDF algorithm.

2.1 Linux-based RTOSes and CPU Scheduling

SCHED_DEADLINE (Faggioli et al, 2009a,b) is an implementation of the EDF algorithm for Linux, which is distributed as a kernel patch tightly integrated with the mainline of the Linux kernel. It also incorporates a CPU reservation mechanism and a multicore scheduler that are respectively based on the well-known CBS and the Global EDF (G-EDF) algorithms. Our CPU scheduler is implemented on top of SCHED_DEADLINE to extend the capability of a CPU scheduler for interactive real-time applications.

LITMUS^{RT} (Calandrino et al, 2006) supports various multicore EDF-based schedulers. It has been used to discover new results on implementation problems related to multicore real-time schedulers (Brandenburg and Anderson, 2009b; Brandenburg et al, 2008). It has also been used to evaluate QoS management (Block et al, 2008) and resource synchronization (Brandenburg and Anderson, 2009c). Currently, its usage is limited to the Intel (x86-32/64) and Sparc64 architectures. In fact, it was reported in (Brandenburg and Anderson, 2009a) that the LITMUS^{RT} project was launched to bridge the gap between research results and industry practice. Meanwhile, the multicore scheduling algorithm supported by our CPU scheduler is aimed at improving CPU bandwidth available for applications with their timing requirements.

Redline (Yang et al, 2008) brings first-class support for interactive applications in commodity OSes. It offers memory and disk I/O management as well as CPU scheduling. The control group mechanism adopted in the mainline of the Linux kernel and the CPU scheduling algorithms provided by SCHED_DEADLINE have nearly covered the Redline techniques. The primary objective of Redline is to ensure available CPU, memory, and disk resources when applications are assigned sufficient CPU bandwidth in the system, while our CPU scheduler 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 RTOSes that exploit CPU reservation include Linux/RK (Oikawa and Rajkumar, 1999), Linux-SRT (Childs and Ingram, 2001), and AQuoSA (Palopoli et al, 2009). 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 soft real-time tasks in the background when their budgets are exhausted, our CPU scheduler 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 depends highly on how a control model is suitable for a system. Meanwhile, our CPU scheduler assigns a 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, our CPU scheduler is aimed at improving the QoS of the overall system and the absolute CPU bandwidth available for applications to perform well.

RTLinux (Yodaiken, 1999), RTAI (Beal et al, 2000), and KURT Linux (Srinivasan et al, 1998) are particularly designed for hard real-time systems. Their requirements

are to minimize the system latency. RED-Linux (Wang and Lin, 1999) 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.

2.2 Memory Management

Some real-time systems are designed to conservatively pin memory pages so that they cannot be swapped out by others (Laplante, 1993). For example, RT-Mach (Tokuda et al, 1990) introduced such a page-pinning mechanism, and it also enabled on-demand page-pinning for interactive applications (Nakajima and Tezuka, 1997). However, as we stated in Section 1, pinning memory pages is expensive, and it often leads to wasting memory resources.

Memory firewalling (Eswaran and Rajkumar, 2005; Hand, 1999) is a reservation-based approach to ensure that a certain number of pages is preserved for the corresponding task, limiting interference to other tasks. In particular, the memory reservation mechanism proposed in (Eswaran and Rajkumar, 2005) was designed for real-time systems and was implemented using the Resource Kernel (RK) abstraction (Rajkumar et al, 1997). Our memory management scheme is inspired by these earlier approaches. However, we propose a more flexible approach that can make interactive real-time applications responsive even though their own memory reserves are exhausted. Specifically, we introduce another type of memory reserve that can be shared by interactive real-time applications. Although the Redline memory management scheme (Yang et al, 2008) also leveraged a concept of shared memory reservation, the size of shared memory reserve is static, and it also does not employ memory firewalling. In contrast, our memory reservation mechanism expands the size of shared memory reserve dynamically on demand to provide better responsiveness for interactive real-time applications, and it also employs memory firewalling to prevent these applications from affecting each other.

Memory management problems in real-time systems are often considered in the programming language domain (Borg et al, 2006; Pizlo and Vitek, 2006; Robertz, 2002), where the Real-Time Specification for Jave (RTSJ) (Gosling and Bollella, 2000) is a primary programming model. Such a language-level support for memory management is, however, not within the scope of this paper, and we only focus on OS-level memory management.

3 System Model

The system is composed of m CPUs, $\pi_1, \pi_2, ..., \pi_m$. Each application task τ_i is a sporadic task, which is characterized by minimum inter-arrival time T_i (also called the period), relative deadline D_i, and worst-case computation time C_i. It sporadically generates a sequence of jobs, each of which must be separated at least by an interval of T_i. A job of τ_i released at time t has a deadline at time $d_i = t + D_i$.

A server with a capacity 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

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 amount of CPU time to be reserved per period.
`set_memsize`	sets the size of memory block to be reserved at the beginning of task execution.
`wait_period`	suspends (sleeps) until the next period.

deadline at time $s_i = t + P_i$. A job of τ_i is scheduled based on 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 (Faggioli et al, 2009a,b; Palopoli et al, 2009; Yang et al, 2008), we associate one server with one task.

A generic hierarchical memory model is adopted. Each CPU has a small L-1 cache. The L-2 cache of relatively large size is shared across all CPUs. The L-3 and L-4 cache could also be accommodated. There is an external storage device to swap pages in to and out from the main memory. We assume that the swap space is large enough to accommodate all swapped-out pages.

Applications use a set of application programming interfaces (APIs) or system calls, supporting 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 RTOSes (Beal et al, 2000; Calandrino et al, 2006; Faggioli et al, 2009a; Oikawa and Rajkumar, 1999; Yang et al, 2008) implement these functions in the kernel, while others (Palopoli et al, 2009; Srinivasan et al, 1998) 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 make the necessary changes in the Linux kernel.

4 CPU Reservation

Interactive real-time applications are often resource-intensive. Hence, CPU reservation is useful to ensure the QoS of those applications. As most EDF-based schedulers with CPU reservation mechanisms (Abeni and Lipari, 2002; Faggioli et al, 2009a,b; Palopoli et al, 2009; Yang et al, 2008), our CPU scheduler 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. See the original papers (Abeni and Buttazzo, 1998, 2004) 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 The New Reservation Algorithm

We propose a new CPU reservation algorithm, called *Flexible CBS* (FCBS). The FCBS algorithm uses a slack-reclaiming approach that has also been considered in previous schedulers (Caccamo et al, 2005; Lin and Brandt, 2005; Marzario et al, 2004). Our approach is however distinguished from these previous work in that (i) it applies exactly 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 for each job even if the reclaiming occurs, and (iii) it reclaims the budget so that the jobs overrunning out of reservation are assigned more CPU bandwidth.

The FCBS algorithm donates the budget left unused by some jobs to either the jobs overrunning out of reservation 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, the schedulability associated with the CBS algorithm is preserved.

The above observation implies that we could improve the average QoS of the overall system, if we efficiently reclaim the budget left unused by some jobs, and donate them to other jobs. We therefore consider the following budget-reclaiming algorithm, coordinating with the CBS algorithm. 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 overrunning out of reservation. 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.

```
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
```

Fig. 3 Pseudo-code of our CPU scheduler with the FCBS algorithm.

2. If there is no such 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 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 of time as consumed by an idle task 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, like the second step described above.

The FCBS algorithm is easily extended to multicore platforms. 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. Meanwhile, if tasks are partitioned among CPUs, the remaining budget can only be used for the tasks on the same CPU. In Section 5, we will discuss how to implement the FCBS algorithm for multi-core platforms.

Implementation. Figure 3 illustrates the pseudo-code of the FCBS algorithm implementation, added to SCHED_DEADLINE, where t_{now} denotes the current time, and t_{last} holds the last time when a job completed. The `budget_exhausted_fcbs`

function is called when the budget of τ_i reaches zero. This function is inserted at the end of the `deadline_runtime_exceeded` function in SCHED_DEADLINE. The `job_released_fcbs` function, meanwhile, is called when a job of τ_i is released. This function 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[1], 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 Scheduling

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 (Faggioli et al, 2009a,b), SCHED_DEADLINE applies a global scheduling policy to the EDF algorithm.

It was shown in (Dhall and Liu, 1978) that the EDF algorithm is no longer optimal on multicore platforms whether we apply partitioned scheduling or global scheduling. Consider $m+1$ tasks on an m-CPU system, all of which require CPU time $P/2+\varepsilon$ every period P, where ε is a small number. Given that the available CPU bandwidth of the system is m, while the total CPU bandwidth required by the $m+1$ tasks is $m(1/2+\varepsilon/P)$, the system should have enough CPU bandwidth. However, this task set is not schedulable in both partitioned scheduling and global scheduling. For partitioned scheduling, it is obvious that the utilization of one CPU exceeds 100%. For global scheduling, on the other hand, assume that all the first jobs of the $m+1$ tasks start at time t. Some m tasks are then dispatched at this time, and their jobs complete at time $t+(P/2+\varepsilon)$. Now, the remaining task misses its deadline, since the laxity to its deadline is only $P/2-\varepsilon$, while it needs to consume CPU time $P/2+\varepsilon$ by its deadline.

Recently, several authors (Andersson et al, 2008; Bletsas and Andersson, 2009; Kato et al, 2009) have developed a new concept: a *semi-partitioned* scheduling approach. Particularly, the Earliest Deadline First with Window-constraint Migration (EDF-WM) algorithm (Kato et al, 2009) 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. We also discuss the CPU assignment problem in the partitioning stage to make the scheduler more robust for reservation-based systems as well as mis-specifications of task parameters.

[1] We refer to the flag names that were given in the January 2010 version of SCHED_DEADLINE. Some of them have different names in the current version.

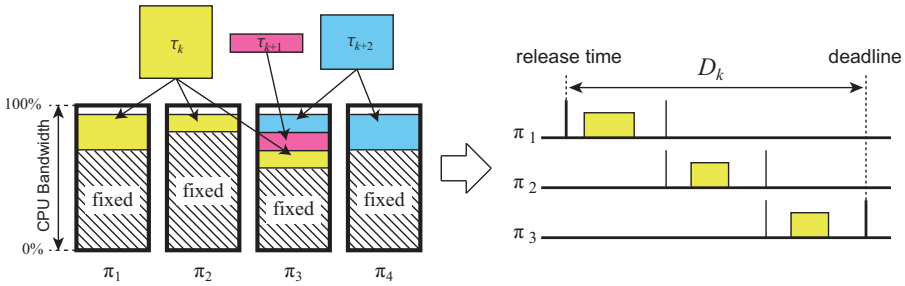

Fig. 4 Concept of semi-partitioned scheduling (Kato et al, 2009).

5.1 The EDF-WM Algorithm

We now briefly describe the EDF-WM algorithm. See (Kato et al, 2009) for details. In the following, we assume that no tasks have reserves for simplicity of description. Specifically, C_i, D_i and T_i are used to schedule each task τ_i.

Figure 4 illustrates a concept of semi-partitioned scheduling presented in previous work (Kato et al, 2009). 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 number of 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. C'_i is now 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 π_x, L is any value equal to the multiple of relative deadline of $\tau_j \in \Gamma_x$, 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 The New Scheduling Algorithm

We now modify the EDF-WM algorithm for our CPU scheduler. The modified algorithm is called *EDF-WM with 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 (Kato et al, 2009). 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 a new task to the first CPU, upon which the schedulability test is passed. In fact, most existing EDF algorithms (Baker, 2006; Bletsas and Andersson, 2009; Lopez et al, 2004) based on partitioned scheduling use the FF heuristic, since it bounds the worst-case schedulability (Lopez et al, 2004). However, in our CPU scheduler, 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 the mis-specifications of task parameters.

The WF heuristic performs load-balancing across all CPUs, while the FF heuristic tries to pack earlier CPUs fully, leaving later CPUs less packed. As a result, under

```
 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
```

Fig. 5 Pseudo-code of our CPU scheduler with the EDF-WMR algorithm.

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 mis-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 heuristic. In Section 7, we show that the WF heuristic performs well for our CPU scheduler.

Implementation. Figure 5 illustrates the pseudo-code of the implementation of the EDF-WMR algorithm, added to SCHED_DEADLINE. Note that the FCBS algorithm has already been implemented. Since the pseudo-code to assign CPUs to a migratory task was already shown in (Kato et al, 2009), 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 respectively appended at the ends of the `budget_exhausted_fcbs`, `job_released_fcbs`, and `job_completed_fcbs` functions that are listed in Figure 3. The `job_migration` function internally uses the `set_cpus_allowed_ptr` func-

tion, 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 Memory Reservation

While CPU scheduling is a *time* allocation problem, memory management is a *space* allocation problem. We propose the Private-Shared-Anonymous Paging (PSAP) algorithm for allocating memory pages in interactive real-time systems. The main goal of the PSAP algorithm is making interactive real-time tasks responsive even under memory pressure, while providing sufficient memory resources to best-effort tasks. The PSAP algorithm separates the physical memory space into several blocks. Each interactive real-time task can create a *private*-reserve block where pages are never stolen by other tasks. There is also a *shared*-reserve block where pages are shared by all interactive real-time tasks but are never stolen by best-effort tasks. The rest of memory space, which is categorized into neither a private- nor a shared-reserve block, is called an *anonymous* memory block where any tasks are allowed to get pages.

6.1 Page Allocation

As the three types of memory blocks imply, interactive real-time tasks and best-effort tasks are allocated pages in different ways. Each interactive real-time task can create a private-reserve block that includes a sufficient number of memory pages for executing the first few jobs, e.g., about 32MB (8000 pages) for an *MPlayer* task performed in our experimental evaluation. We also suggest that system designers allocate a similar number of memory pages as shared-reserve pages.

Interactive real-time tasks first attempt to get pages from their own private-reserve blocks, if any. If there are no free pages in these memory blocks, they next try to get pages from the shared-reserve block, competing with other interactive real-time tasks. If this memory block is also exhausted, they seek *expired* (or inactive) pages in their own private-reserve blocks, and reclaim these pages. Our page reclamation policy *steals* pages from the anonymous memory block into the shared memory block to supply additional pages for interactive real-time tasks. See Section 6.2 for a description of the page expiration policy and Section 6.4 for a description of the page reclamation policy. If expired pages are still not found, they finally reclaim their *active* pages based on a least-recently-used (LRU) policy, contending with themselves.

In contrast, best-effort tasks are allowed to get pages only from the anonymous block. They are simply allocated pages from this memory block in accordance with a traditional LRU-based paging policy.

Implementation: Our implementation simply reuses the `alloc_page` and the `__free_page` functions provided by the Linux kernel. Specifically, we get the reserved pages through the `alloc_page` function when tasks start, and free them through the `__free_page` function when tasks exit. These functions itself are also modified a bit. The `alloc_page` function first tries to allocate pages from its private-reserve

block or the shared-reserve block, if the current task is running under memory reservation. Otherwise, it proceeds with the original procedure, i.e., allocating pages from the anonymous block. The `__free_page` function also checks if the target page is part of the reserve blocks. If so, it does not really free the page but keeps it in the reserve blocks as a free page. Otherwise, it proceeds with the original procedure.

6.2 Page Expiration

We do not prefer to *pin* pages in physical memory for interactive real-time tasks, since some previously-used pages are not accessed again, being left unfreed, until the tasks exit. In fact, many such pages are observed in executing *MPlayer*. Meanwhile, we are also not willing to apply an LRU policy to interactive real-time tasks, since their pages may easily be reclaimed and swapped out to an external storage device by best-effort tasks accessing memory frequently. We therefore focus on the length of time during which the pages of interactive real-time tasks are not accessed. If the pages are not accessed for a certain duration, these pages are marked as *expired* pages that can be reclaimed, even though they are dirty. The length of this duration is configurable. In our implementation, we set the length $10 \times T_i$ for each interactive real-time task τ_i, meaning that its pages can be swapped out if they are not accessed while executing 10 successive jobs. We expect that such pages would not be accessed again.

6.3 Reserve Expansion

The private-reserve pages ensure that the corresponding interactive real-time task can get *at least* the reserved number of pages for itself, and these pages are not used by any other jobs. As mentioned above, we recommend that interactive real-time tasks have private-reserve pages that would be sufficient for serving only their first few jobs. It is over-provisioning to reserve all pages that will be requested by interactive real-time tasks, since some tasks may allocate new pages in every period, while not freeing old pages at all, like *MPlayer*. However, an under-provisioning reservation could force interactive real-time tasks to exhaust their private-reserve pages, even if some pages are reclaimed by the above page expiration policy.

The shared-reserve block is useful under this circumstance. It provides pages to interactive real-time tasks that have exhausted their private-reserve pages. When some tasks get pages from the shared-reserve block, the shared-reserve block *steals* the same number of pages from the anonymous block. In other words, it *expands* dynamically. Thanks to this reserve expansion, we can likely keep as many free pages as possible in the shared-reserve block, thus preventing interactive real-time tasks from causing pages to be swapped. When the pages allocated from the shared-reserve block are freed, the PSAP algorithm shrinks the shared-reserve block by the same size as the freed pages. It should be noted that the PSAP algorithm creates another very small space (currently set to be $1/2500$ of the entire main memory space) as part of the anonymous block, which can only be used by best-effort tasks, in order to prevent them from starving.

The shared-reserve block is useful not only when interactive real-time tasks exhaust their private-reserve pages but also when these tasks start execution in the face of high memory pressure. Specifically, if there are no free pages in the anonymous block when these tasks try to create their private-reserve blocks, the PSAP algorithm allows them to steal the required size of memory block from the shared-reserve block that will expand later. Page reclamation may eventually be forced, if the required size is larger than the size of the shared-reserve block.

6.4 Page Reclamation

In the PSAP algorithm, there are three possible scenarios where interactive real-time tasks trigger page-swapping. The most likely scenario is that there are no free pages in the anonymous block when the shared-reserve block tries to expand. The second scenario is that when some interactive real-time task attempts to get pages, both its private-reserve and the shared-reserve blocks are filled, and thereby needs to reclaim its expired or active pages. The last scenario is that when some interactive real-time task undertakes to create a private-reserve block, there are no free pages in the anonymous block, and the shared-reserve block is also not large enough.

Page reclamation could force pages to be swapped, which results in causing tasks to spend additional time on slow disk accesses. We therefore prevent interactive real-time tasks from directly invoking page-swapping procedures, unless they need to allocate pages immediately. Specifically, we focus on the case where the shared-reserve block tries to expand when there are no free pages in the anonymous block. We need some pages to be reclaimed from the anonymous block, and these pages may be swapped out. However, we do not prefer page-swapping to be triggered by interactive real-time tasks, since they will be blocked until new pages are swapped in. Hence, in our page reclamation mechanism, we just set a flag to indicate that we want new pages. Every time the CPU scheduler is invoked to switch tasks, it checks the flag *after* a context switch takes place. If the flag is set, and the current (new) task is a best-effort task, then we try to expand the shared-reserve block. The blocking time is thus imposed on this best-effort task.

Our page reclamation mechanism is reasonable in two ways. First, it is less likely that interactive real-time tasks exhaust free pages in the anonymous block *at once*, as they often request a proper size of memory block in each frame processing. Therefore, page reclamation for expanding the shared-reserve block can usually defer for some interval to trigger when best-effort tasks are dispatched. Second, most interactive real-time tasks have self-suspension times due to I/O blocking, meaning that interactive real-time tasks are likely to yield CPU times to best-effort tasks. This fact is important in that page reclamation under the PSAP algorithm should occur when best-effort tasks are dispatched.

7 Evaluation

We now present a quantitative evaluation of our CPU scheduler and memory management scheme. Our evaluation platform contains a 2.0 GHz Intel Core 2 Quad proces-

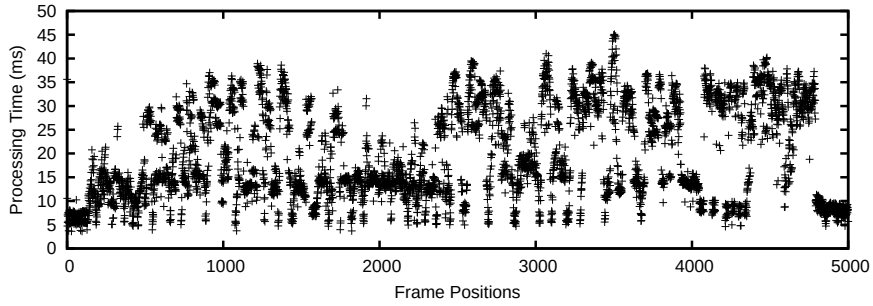

Fig. 6 Frame processing time of the experimental H.264 movie.

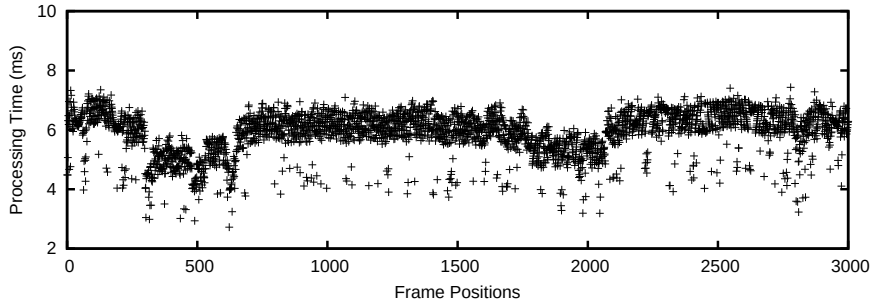

Fig. 7 Frame processing time of the experimental uncompressed movie.

sor (Q9650), 2 GB of DDR2-SDRAM, and an Intel GMA 4500 integrated graphics card. The Linux 2.6.32 kernel using the SCHED_DEADLINE patch is used as the underlying OS.

Movie Player: The MPlayer open-source movie player is used to demonstrate the representative workload of integrative real-time applications. In order to make MPlayer available with interface of our CPU scheduler and memory management scheme listed in Table 1, its source code is slightly modified, but the modification is limited to only 6 lines of code and one additional header file. The MPlayer task executes as a real-time task in the Linux kernel so that it is not affected by other background activities.

The MPlayer task plays two types of movies. One is compressed by the *H.264* 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 large differences among the average-case, the best-case, and the worst-case execution times. The frame processing time of this movie achieved by the MPlayer task is demonstrated in Figure 6. 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, as its frame processing time is shown in Figure 7. Since the frames are not compressed, the processing times do not fluctuate as much as the H.264 movie.

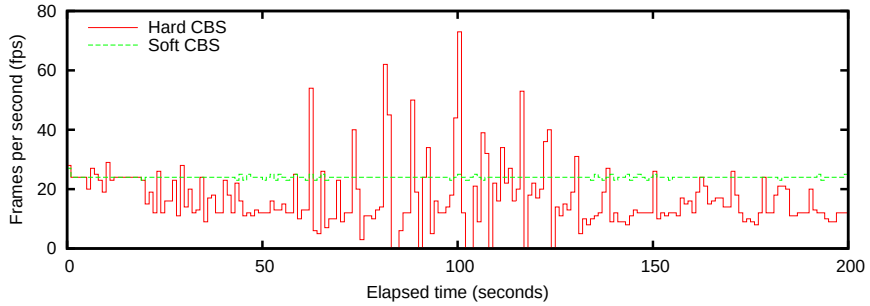

Fig. 8 Frame-rate of the H.264 movie under two variants of the CBS algorithm.

CBS variants: It should be noted that there are two variants of the CBS algorithm, as discussed in (Abeni and Buttazzo, 2004). When we reserve CPU time Q per period P for some task, one policy that we refer to as a *hard* reservation policy allows the task to utilize CPU bandwidth of strictly Q/P, while the other policy that we refer to as a *soft* reservation policy allows *at least* Q/P. Figure 8 shows the frame-rate of the H.264 movie, whose processing time is demonstrated in Figure 6, achieved by the MPlayer task under these two variants of the CBS algorithm implemented using SCHED_DEADLINE. We set 25*ms* to be the budget, given that the worst-case execution time of around 45*ms* is too pessimistic for reservation in this case, and it is rather appropriate to reserve the intermediate processing time per period. According to these results, the soft CBS policy ensures stable QoS for the movie playback, whereas the hard CBS policy repeatedly *drops and catches up* frames. This preliminary evaluation suggests that we use the soft CBS policy for interactive real-time applications with variable execution times. In fact, SCHED_DEADLINE now uses the soft CBS policy, while it used to adopt the hard CBS policy. Our evaluation provided herein uses this new version of SCHED_DEADLINE, using the *soft* CBS algorithm.

7.1 CPU Scheduling

We first evaluate the capabilities of our CPU scheduler to execute multiple MPlayer tasks playing the H.264 movie. As observed in Figure 6, a majority of frames are processed within 25*ms*. We hence set 25*ms* to be the CPU time reserved per period for both our CPU scheduler and SCHED_DEADLINE, in order to ensure the QoS of this majority. As for our CPU scheduler, we prepare three configurations to see how the FCBS and the EDF-WMR algorithms improve the frame-rate respectively. The first configuration uses only the FCBS algorithm, the second uses only the EDF-WMR algorithm, and the last uses both the FCBS and the EDF-WMR algorithms. When the FCBS and/or the EDF-WMR algorithms are disabled, our CPU scheduler relies on SCHED_DEADLINE to reserve CPU bandwidth and/or to migrate tasks.

Figure 9 depicts the worst number of frames per second obtained by playing 4 instances of the H.264 movie concurrently. Since we pick the worst number among all the movies at every sampling point, the results do not correspond to particular movies

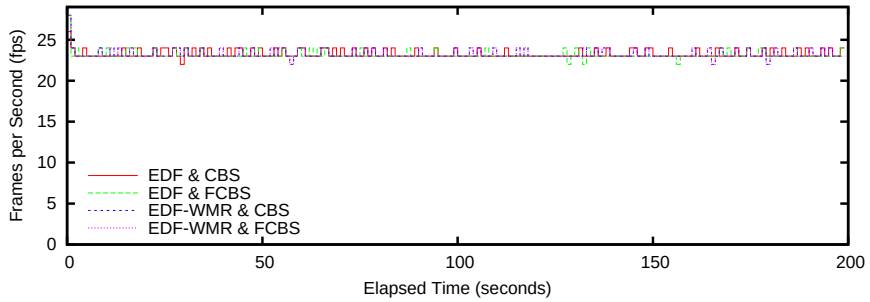

Fig. 9 The worst-case frame-rate in 4 instances of the H.264 movie.

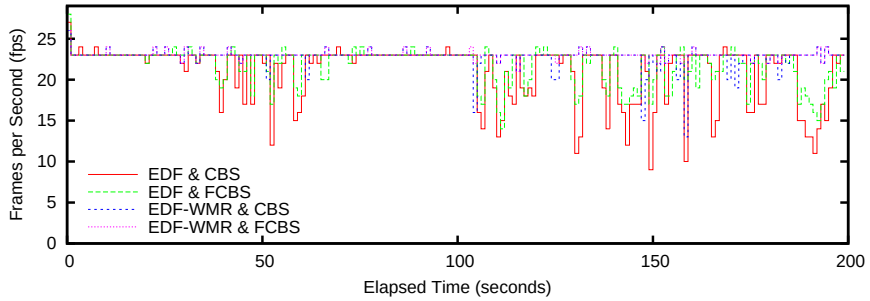

Fig. 10 The worst-case frame-rate in 5 instances of the H.264 movie.

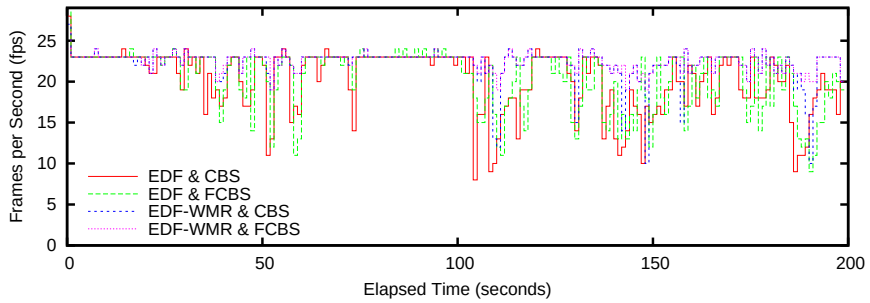

Fig. 11 The worst-case frame-rate in 6 instances of the H.264 movie.

but indicate the QoS of the overall system. In this setup, all CPU schedulers achieve stable frame-rates. Since we have four CPUs available in a quad-core machine, one CPU is exclusively assigned to each MPlayer task. Therefore, the frame-rates are kept stable. However, the frame-rates occasionally drop below 24 fps, since some individual frames require the processing time more than the period, as shown in Figure 6, since the frame processing time is affected by the resource usage, like a cache status, which is not predictable due to the existence of Linux background jobs.

Figure 10 shows the worst-case frames per second obtained by playing 5 instances of the H.264 movie concurrently. In this setup, our CPU scheduler outper-

forms SCHED_DEADLINE. Particularly, when both the EDF-WMR and the FCBS algorithms are used, the frame-rate is maintained at nearly 24 fps. Given that there are 5 MPlayer tasks, each of which reserves CPU bandwidth of $25/41 = 61\%$, they apparently need to migrate across CPUs to meet deadlines. In this scenario, the timely migration mechanism provided by the EDF-WMR algorithm correctly allocates the CPU time to each task, while the EDF algorithm implemented in SCHED_DEADLINE causes the tasks to miss deadlines more frequently. Hence, we conclude that a semi-partitioned multicore scheduling approach is effective to support QoS-aware real-time applications. On the other hand, using only one of the FCBS and the EDF-WMR algorithms causes the frame-rate to drop a little. It is however worth noting that these configurations still provide better frame-rates than SCHED_DEADLINE. Therefore, both the FCBS and the EDF-WMR algorithms are useful to improve the QoS of the overall system over the CBS and the EDF algorithms respectively.

Figure 11 shows the worst-case frames per second obtained by playing 6 instances of the H.264 movie concurrently using MPlayer. In this setup, even our CPU scheduler using both the FCBS and the EDF-WMR algorithms face degradation in the frame-rate, but still provides the frame-rate of over 20 fps, whereas the frame-rate is severely affected by each other in other configurations. In fact, 6 periodic tasks with computation time 25*ms* and period 41*ms* are theoretically schedulable by the EDF-WM algorithm. Furthermore, even though their jobs spend more than 25*ms* of CPU time, the FCBS algorithm covers up their overruns. As a consequence, these 6 MPlayer tasks are kept from any fatal timing violation under our CPU scheduler using both the FCBS and the EDF-WMR algorithms.

In our experience, when frame-rates drop below 20 fps, we visually notice that the movies are delayed. When the frame-rates drop below 15 fps, we even start feeling uncomfortable watching the movies. In this regard, our CPU scheduler is useful to provide interactive real-time applications with sufficient QoS.

We next evaluate the average-case frame-rate and the associated standard deviation rather than the worst-case bursts and drops in the frame-rate. We particularly focus on the case that has demonstrated the most burst interference in the previous experiments, where 6 MPlayer tasks execute concurrently, as shown in Figure 12. Interestingly, the average-case frame-rate is not very different among all the configurations under consideration. The standard deviation, however, indicates that the frame-rate is not stable under SCHED_DEADLINE, while it becomes more stable as our CPU scheduler uses the FCBS algorithm and/or the EDF-WMR algorithm. Thus, our CPU scheduler is beneficial for not only improving the worst-case frame-rate but also providing stable frame-rates on average.

Figure 13 shows the average-case system processing time taken to compute a single frame of the H.264 movie. It is measured by the `stime` member in the Linux task control block. Since the difference among all the configurations is no more than $15\mu s$, we evince that the additional implementation overhead of our CPU scheduler is negligibly small as compared to SCHED_DEADLINE. The fact that the absolute system overhead increases as the number of concurrently-playing movies increases implies that the scheduling decision spends more CPU time to schedule a larger number of tasks, but the amount of increase is trivial. The implementation of the EDF-WMR algorithm even reduces the system overhead in some case, because each local sched-

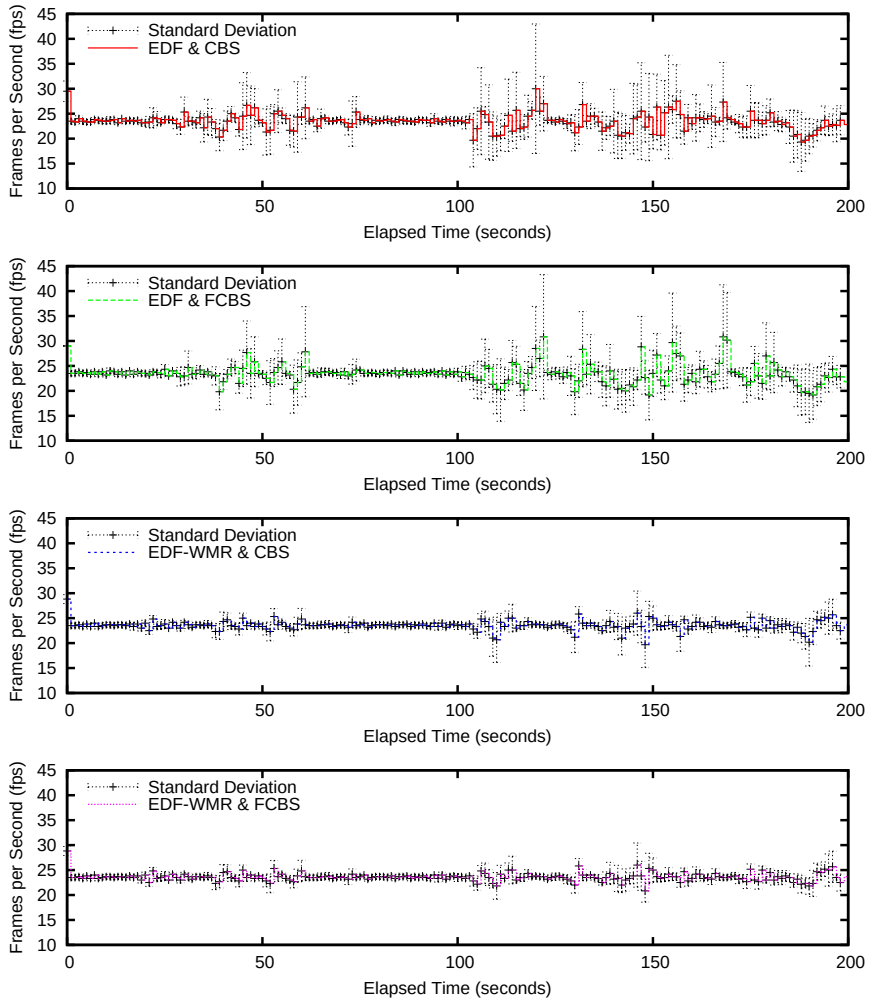

Fig. 12 The average-case frame-rate in 6 instances of the H.264 movie.

uler only needs to manage the tasks assigned to the corresponding CPU under the EDF-WMR algorithm based on semi-partitioned scheduling. As a result, scheduling decision needs less CPU time than SCHED_DEADLINE based on global scheduling.

Figure 14 shows the average frame processing times of the H.264 movie. We observe that the average frame processing times are almost the same in all cases. This means that memory caches affect all these 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 our CPU scheduler in playing multiple instances of the uncompressed movie. As shown in Figure 7, the frame processing time

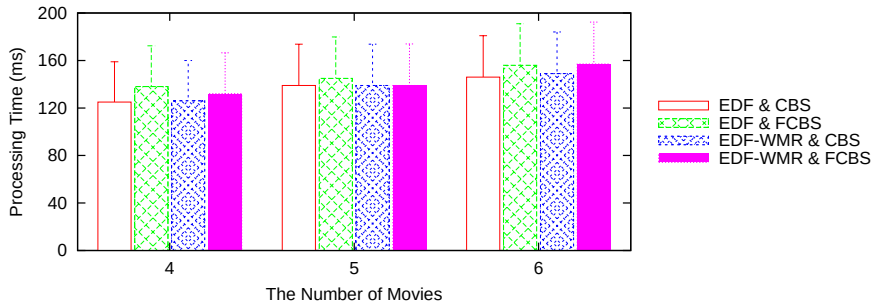

Fig. 13 The average-case system processing time in playing the H.264 movie.

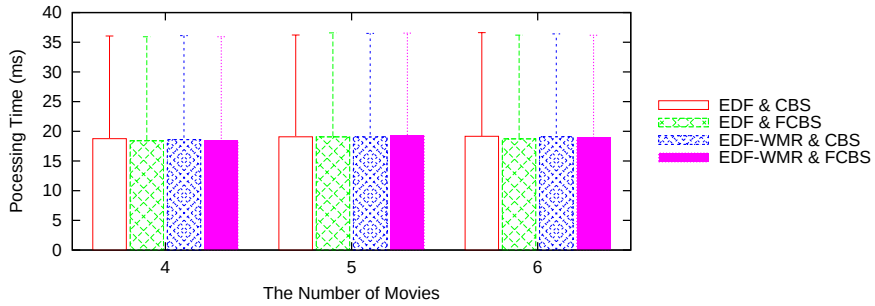

Fig. 14 The average-case user-mode processing time in playing the H.264 movie.

does not fluctuate as much as the H.264 movie, since frames are not compressed. In executing the MPlayer task for this uncompressed movie, we set $9ms$ to be the CPU time reserved per period, which is enough to ensure the QoS of the overall system.

Figure 15, 16, and 17 show the worst number of frames per second obtained by playing multiple instances of the uncompressed movie concurrently. Since the frame processing time is much smaller than that of the H.264 movie, the system can accommodate more movies. Given that the CPU bandwidth required by each MPlayer task is about $9/33 \simeq 27\%$, the system should have enough CPU bandwidth when playing no more than fourteen movies ($9/33 \times 14 < m = 4$). Unlike the case for playing H.264 movies, there is not a large difference in the frame-rate among the three configurations of our CPU scheduler and SCHED_DEADLINE. 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, our CPU scheduler still provides benefits in the frame-rate, when playing more than 12 movies, mainly due to the advantage of the EDF-WMR algorithm over the EDF algorithm. As a consequence, the timely migration mechanism provided by the EDF-WMR algorithm is effective to ensure the QoS of the overall system, even when the frame processing time does not vary a lot.

Figure 18 demonstrates the average-case frame-rate and the associated standard deviation when playing 14 instances of the uncompressed movie. The average-case frame-rate is not very different among our CPU scheduler and SCHED_DEADLINE.

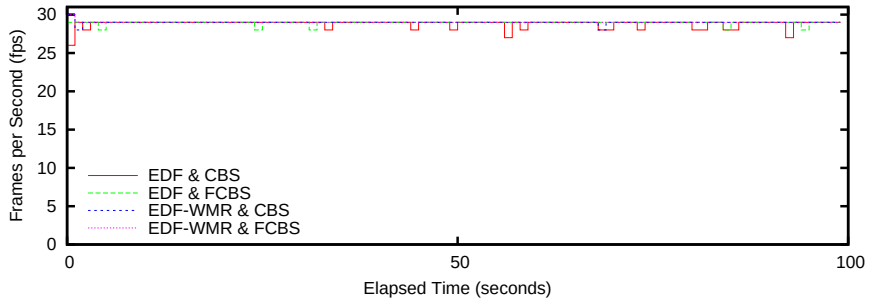

Fig. 15 The worst-case frame-rate in 12 instances of the uncompressed movie.

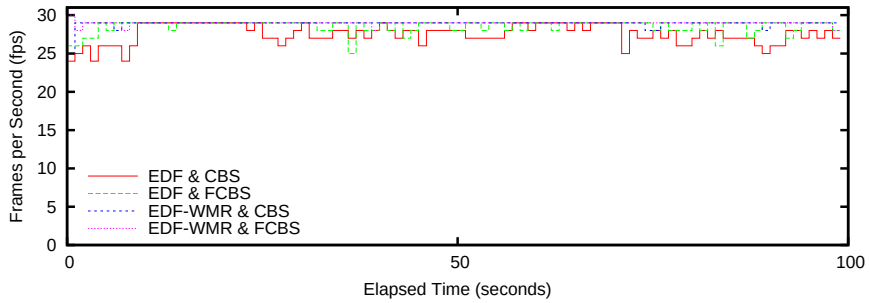

Fig. 16 The worst-case frame-rate in 13 instances of the uncompressed movie.

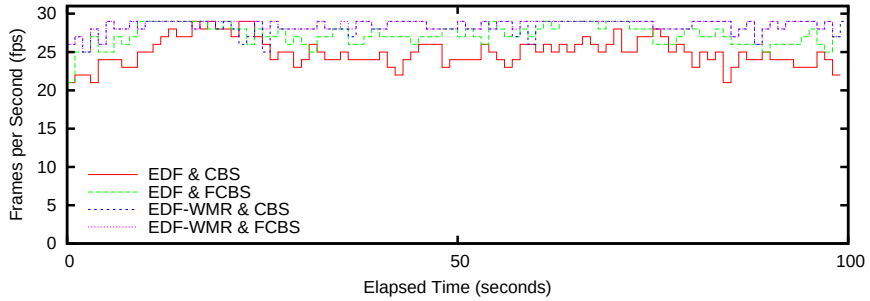

Fig. 17 The worst-case frame-rate in 14 instances of the uncompressed movie.

However, as compared to those for the H.264 movies depicted in Figure 12, the errors from the average-case frame-rate are quite limited, though SCHED_DEADLINE still faces variability in the frame-rate. Our CPU scheduler retains such variability, especially when both the FCBS and the EDF-WMR algorithms are leveraged. Thus, it provides a considerable benefit for playing multiple uncompressed movies as well as H.264 movies. It is interesting to observe that the errors from the average-case frame-rate chronically appear on a time, while the range of error value is limited. Since the frame processing time for the uncompressed movie is more constant than

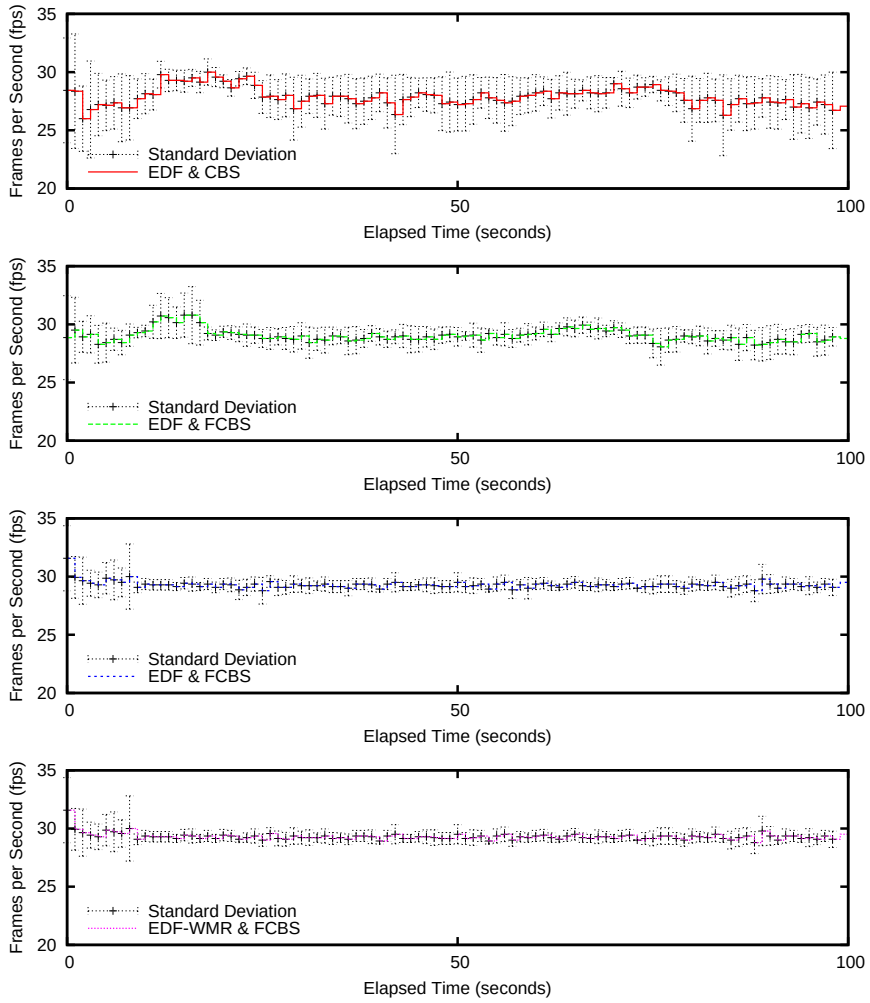

Fig. 18 The average-case frame-rate in 14 instances of the uncompressed movie.

that for the H.264 movie, occasional bursts and drops in the frame-rate are reduced, but instead the frame-rate constantly exhibits some variability.

Figure 19 shows the average-case system processing time taken to compute a single frame of the uncompressed movie. The absolute system processing time is greater than the previous the case for the H.264 movie shown in Figure 13. The increased portion in the system processing time mostly includes scheduling overhead, since there are more than 12 tasks to schedule. However, we claim that the difference between our CPU scheduler and SCHED_DEADLINE is still ignorable.

Figure 20 shows the average-case user-mode processing times of the uncompressed movie. Surprisingly, our CPU scheduler needs the most CPU time to compute a single frame in spite of limited migrations. Even if the EDF-WMR algorithm is not

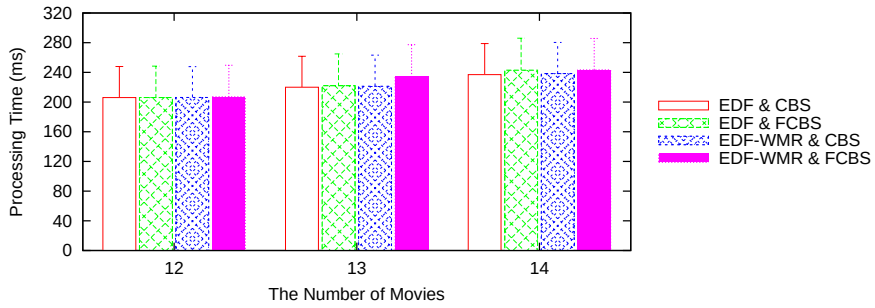

Fig. 19 The average-case system processing time in playing the uncompressed movie.

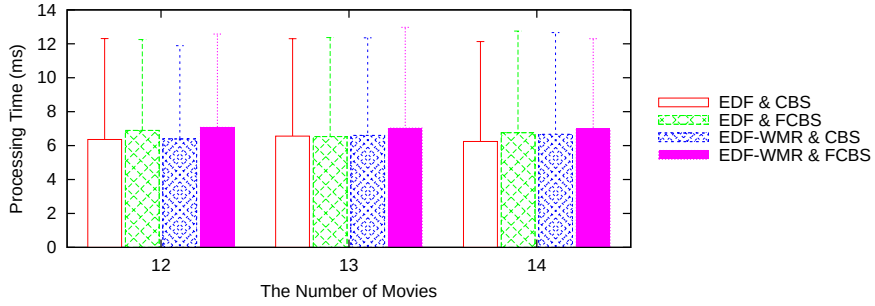

Fig. 20 The average-case user-mode processing time in playing the uncompressed movie.

used, our CPU scheduler needs more processing time than SCHED_DEADLINE. We reason this fact 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 memory caches affect the processing times due to a number of context switches, unlike the previous cases scheduling no more than six tasks. In particular, our CPU scheduler using both the FCBS and the EDF-WMR algorithms assigns particular CPUs to the tasks. The FCBS algorithm is therefore applied independently on each CPU. As a result, the cache performance is affected more. However, our CPU scheduler still improves the frame-rate.

Experiments with busy-loop tasks. We now assess the capabilities of our CPU scheduler in terms of the breakdown workload for hard-deadline guarantee, in order to show that our CPU scheduler can not only maintain the desired frame-rates of applications for soft real-time setup, but also meet the deadlines of applications for hard real-time setup, where tasks have more constant execution times than interactive tasks. Like prior work (Brandenburg and Anderson, 2009b; Brandenburg et al, 2008; Calandrino et al, 2006), we submit many sets of randomly-generated busy-loop periodic tasks to the system, and observe the ratio of task sets successfully scheduled without missing deadlines. The busy-loop tasks used in this evaluation do not access memory to prevent them from affecting execution times. Evaluation for memory-intensive hard-deadline tasks is left open for future work. In the following

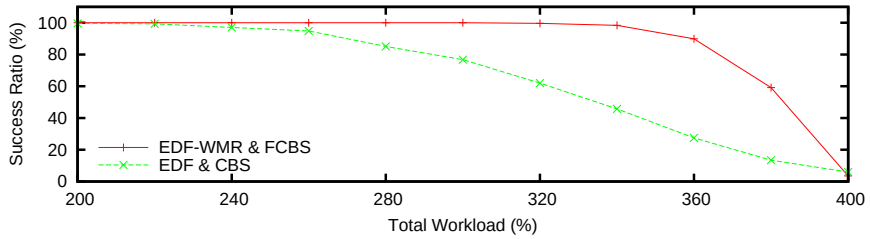

Fig. 21 Hard timing guarantees in scheduling busy-loop tasks with individual utilization $[10\%s, 100\%]$.

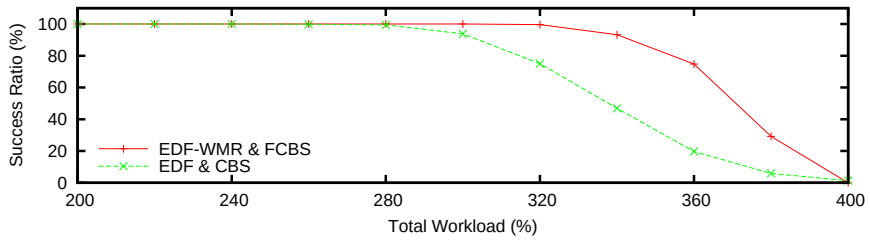

Fig. 22 Hard timing guarantees in scheduling busy-loop tasks with individual utilization $[50\%s, 100\%]$.

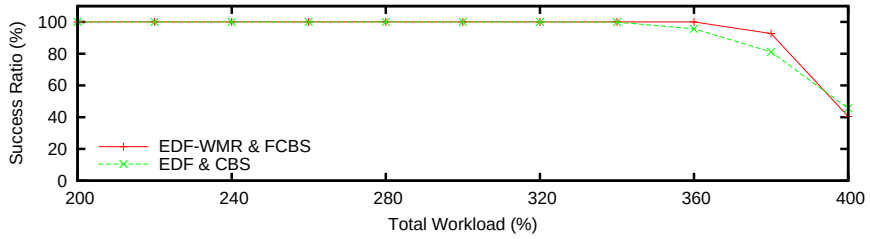

Fig. 23 Hard timing guarantees in scheduling busy-loop tasks with individual utilization $[10\%s, 50\%]$.

experiments, CPU reservation mechanisms are not used since execution times are tightly bounded due to busy-loop tasks.

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 newly-generating task τ_i is determined based on a uniform distribution. The range of the distribution is parametric. In our evaluation, we have three test cases: $[10\%, 100\%]$ (both heavy and light tasks), $[10\%, 50\%]$ (only light tasks), and $[50\%, 100\%]$ (only heavy tasks). 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 $[1ms, 100ms]$. The execution time of τ_i is set to $C_i = U_i T_i$.

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 execute these busy-loop tasks for 10 minutes. A task set is said to be successfully scheduled if and only if all jobs complete within their periods during the measurement. We

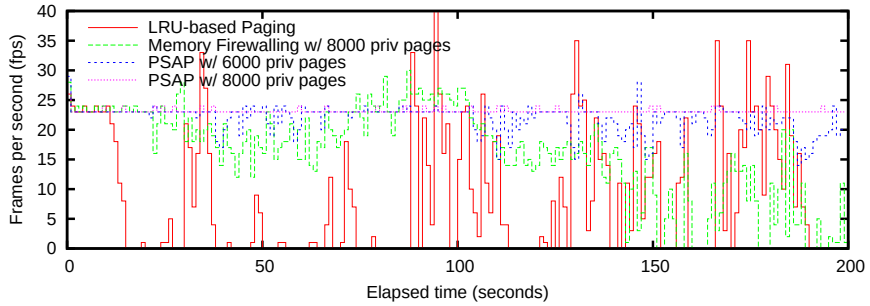

Fig. 24 The worst-case frame-rate in 4 instances of the H.264 movie under memory pressure.

then evaluate by success ratio: *the ratio of the number of successfully-scheduled task sets and that of submitted task sets*.

Figure 21, 22, and 23 show the success ratios in scheduling the task sets of different workloads. We note again that each sampling point runs 1000 task sets. According to the results, our CPU scheduler has an advantage in meeting hard timing guarantees, beyond providing soft real-time performance. It is known (Dhall and Liu, 1978) that the EDF algorithm shows worse schedulability when high-utilization tasks exist. Our experimental results truly reflect this effect.

7.2 Memory Management

We now evaluate our memory management scheme. Specifically, we look into the responsiveness of the MPlayer task in the face of memory pressure. To this end, we create a *memory bomb* task that allocates 500 MB of heap memory, and keeps writing to each page in an infinite loop. We launch four instances of this memory bomb task to burden all space (2 GB) of memory, when we execute the MPlayer task. While we use our CPU scheduler introduced in this paper employing the EDF-WMR and the FCBS algorithms, our memory management scheme can generally be used with any CPU schedulers. We primarily focus on the capability of memory management schemes to protect interactive real-time tasks, i.e., the MPlayer task in this case, from the interference introduced by background memory-intensive tasks. The impact on the performance of such background best-effort tasks could also be of interest to discuss, but we consider that it is out of the scope of this paper.

Figure 24 shows the worst-case frame-rate obtained by playing 4 instances of the H.264 movie under the three memory management schemes. For the memory firewalling algorithm, we reserve 8000 pages for each MPlayer task, which should be enough for executing the first few frames of the H.264 movie. Meanwhile, we prepare two configurations for the PSAP algorithm to see the impact as a function of different sizes of memory reserves. For each MPlayer task, the first configuration reserves 8000 pages, while the second reserves 6000 pages. In either case, 8000 pages are reserved for the shared-reserve block. As depicted in Figure 9, when there is no memory pressure, the frame-rates of 4 MPlayer tasks are all stable. However,

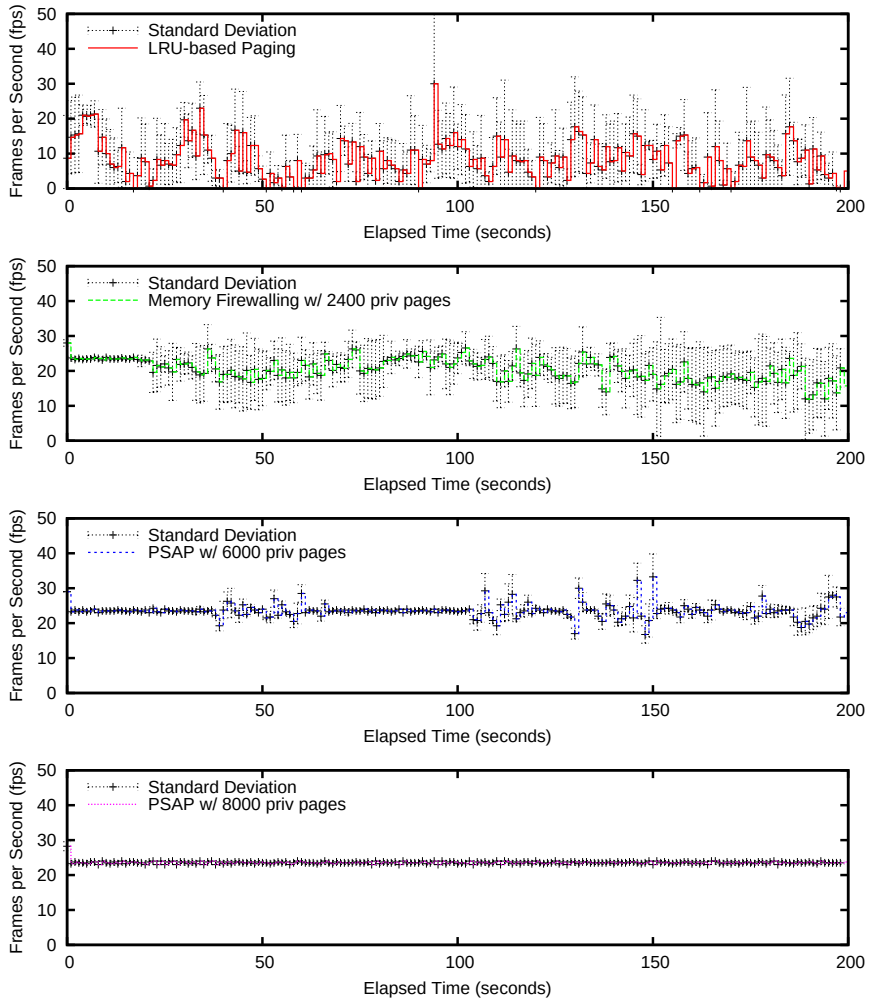

Fig. 25 The average-case frame-rate in 4 instances of the H.264 movie under memory pressure.

these MPlayer tasks are significantly affected by memory bomb tasks under the traditional LRU-based paging algorithm, which is adopted in the Linux kernel by default. The memory firewalling algorithm (Eswaran and Rajkumar, 2005) achieves better frame-rates for MPlayer tasks, but it still decreases frame-rates to about 15 fps in the worst case. This is largely attributable to the fact that MPlayer tasks are occasionally blocked due to page-swapping invoked by themselves, when pages are exhausted in their own reserved memory block. On the other hand, the PSAP algorithm successfully isolates MPlayer tasks from memory pressure. Even though pages are exhausted in their own private memory blocks, the tasks can get pages from the shared-reserve block. It can therefore maintain frame-rates to be almost ideal around 24 fps. Although the memory firewalling algorithm could perform better if we assign

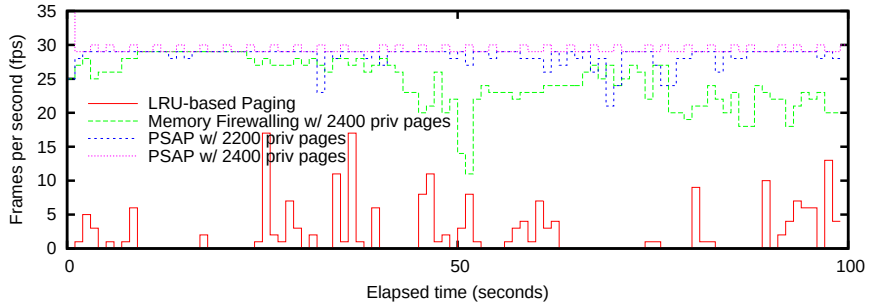

Fig. 26 The worst-case frame-rate in 12 instances of the uncompressed movie under memory pressure.

a large amount of memory reserve for each MPlayer task, such an over-provisioning approach may waste memory resources, and degrade the performance of best-effort tasks. It is also very interesting to see that even when the size of private memory block is reduced to 6000 pages in the PSAP algorithm, it still provides better frame-rates than the memory firewalling algorithm where 8000 pages are allocated for each task as a private memory block. Since the PSAP algorithm also uses 8000 pages for the shared memory reserve, this result indeed evinces that the PSAP algorithm outperforms the memory firewalling algorithm when the same total size of memory reserve is assigned, i.e., 32000 pages in this setup.

Figure 25 shows the average-case frame-rate and the associated standard deviation in playing 4 instances of the H.264 movie. In this setup, even the average-case frame-rate is very different among the PSAP, memory firewalling, and LRU-based paging algorithms. When the PSAP and the memory firewalling algorithms use the same total size of memory reserve by 32000 pages, both the average-case frame-rate and the errors are much more stable under the PSAP algorithm. When they use the same size of private memory reserve by 32000 pages, the PSAP algorithm provides further better frame-rates due to the existence of the shared memory reserve. Therefore, the shared memory reserve and the page reclamation mechanism provided by the PSAP algorithm play a vital role in protecting interactive real-time applications from the interference introduced by memory pressure.

Figure 26 shows the worst-case frame-rates of concurrently-executing 12 instances of MPlayer, playing the uncompressed movie under the three memory management schemes. In this setup, we reserve 2400 pages for each MPlayer task, which should be enough for executing the first few frames of the uncompressed movie, when we use memory reservation mechanisms. The shared-reserve block also includes the same number of pages, when we use the PSAP algorithm. The obtained results are similar to those for playing the H.264 movies. Therefore, it is demonstrated that the PSAP algorithm is effective under various configurations.

The average frame-rate in playing 12 uncompressed movies and the associated standard deviation are shown in Figure 27. The LRU-based paging algorithm severely suffers from the interference introduced by memory pressure even for the average performance. The memory firewalling algorithm can somewhat protect the MPlayer tasks by reserving the private memory block, but the frame-rate still varies. The PSAP

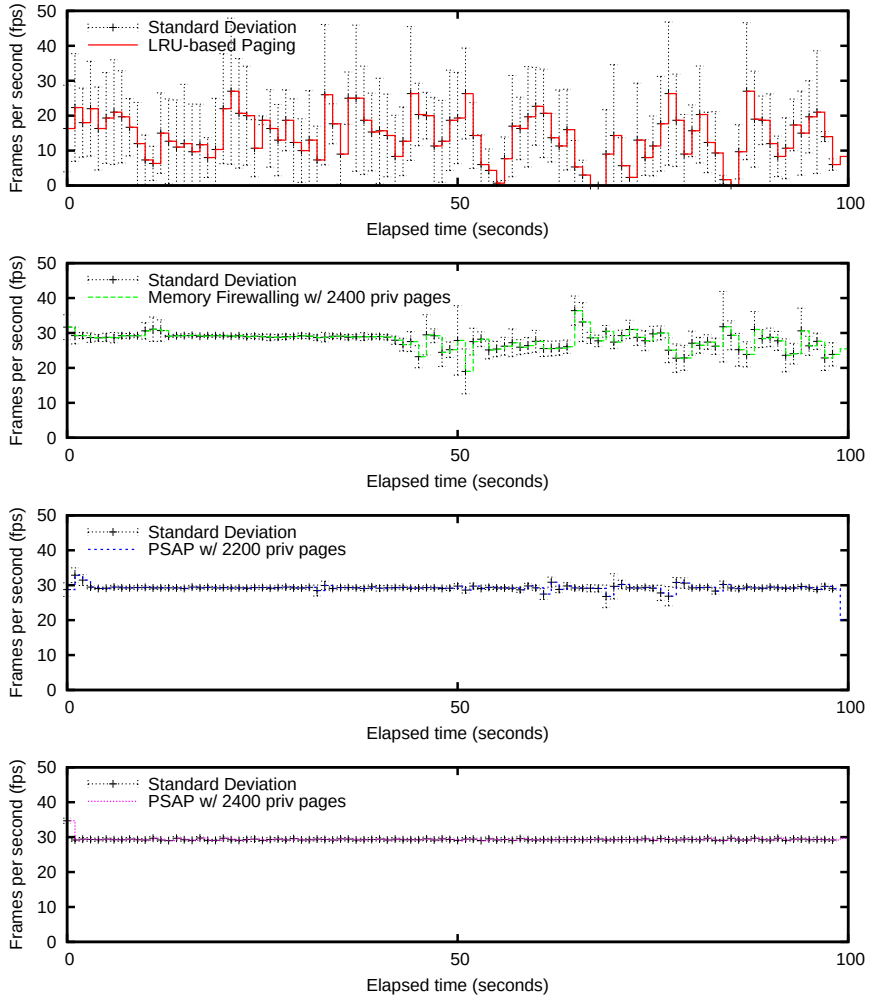

Fig. 27 The average frame-rate in 12 instances of the uncompressed movie under memory pressure.

algorithm, on the other hand, correctly protects the MPlayer tasks from the memory interference. Considering the same advantage observed in the previous experiments, the PSAP algorithm is effective for various workloads.

Figure 28 depicts the average-case processing time imposed on the system in executing the MPlayer task. It is measured by the `stime` member of the Linux task control block. While the PSAP algorithm could take a longer time to do paging itself than memory than the memory firewalling and LRU-based paging algorithms, it reduces the number of paging processes imposed on the MPlayer task running as an interactive real-time task. As a result, the total system processing time observed in the MPlayer task is smaller in the PSAP algorithm than in the memory firewalling and LRU-based paging algorithms.

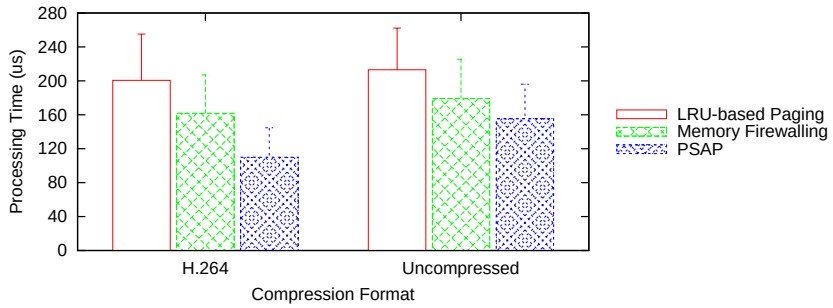

Fig. 28 The average-case system processing time in playing the H.264/uncompressed movie.

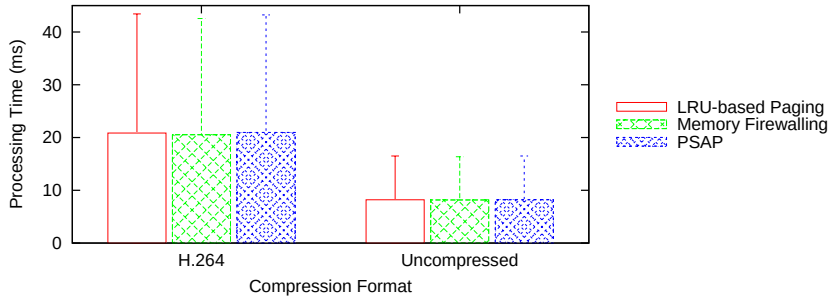

Fig. 29 The average-case user-mode processing time in playing the H.264/uncompressed movie.

Figure 29 shows the average-case processing time of the MPlayer task in the user mode. It is measured by the `utime` member of the Linux task control block. There is little difference in the user-mode processing time among the PSAP, memory firewalling, and LRU-based paging algorithms, since the major penalty in page-swapping is a disk access, while it cannot be accounted by the `stime` and `utime` members. The actual processing time including the disk access, however, should be different among these algorithms, as we have observed the impacts on the frame-rate through Figure 24 to Figure 27. In fact, Figure 28 implies that the memory firewalling and LRU-based paging algorithms cause more page-swapping while the MPlayer task is executing than the PSAP algorithm, which resulted in significant impacts on the frame-rate of the MPlayer task.

8 Concluding Remarks

We have presented the design and implementation of a CPU scheduler and a memory management scheme for interactive real-time applications. Our CPU scheduler 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 imple-

mented using the SCHED_DEADLINE patch (Faggioli et al, 2009a,b) made for the Linux kernel. We also have presented a memory reservation mechanism that uses a new paging algorithm, called PSAP.

Our detailed experimental evaluation showed that our CPU scheduler 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 our CPU scheduler is negligibly small. It also showed that our CPU scheduler is able to improve the breakdown workload for hard timing guarantees, when running randomly-generated task sets, as compared to SCHED_DEADLINE. In addition, we demonstrated that our memory management scheme is able to protect interactive real-time tasks from memory pressure, while these tasks inevitably have lower frame-rates without our memory management scheme.

In future work, we will consider the *coordination* of computing resource management, not just CPU scheduling and memory management, but also network scheduling and I/O management. We believe that there are many types of interactions among these resource management techniques, which need to be investigated for interactive real-time applications. We are also interested in implementing many existing resource management algorithms to conclude what algorithms are best suited for interactive real-time applications. The most challenging issue is how to find optimal configuration for the reservation capacity/period in the FCBS algorithm and the number of reserved pages in the PSAP algorithm.

References

Abeni L, Buttazzo G (1998) Integrating multimedia applications in hard real-time systems. In: Proc. of the IEEE Real-Time Systems Symposium, pp 3–13

Abeni L, Buttazzo G (2004) Resource reservation in dynamic real-time systems. Real-Time Systems pp 123–167

Abeni L, Lipari G (2002) Implementing resource reservations in Linux. In: Real Time Linux Workshop

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

Andersson B, Bletsas K, Baruah S (2008) Scheduling arbitrary-deadline sporadic task systems on multiprocessors. In: Proc. of the IEEE Real-Time Systems Symposium, pp 385–394

Baker T (2006) 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

Beal D, Bianchi E, Dozio L, Hughes S, Mantegazza P, Papacharalambous S (2000) RTAI: Real Time Application Interface. Linux Journal 29:10

Bletsas K, Andersson B (2009) Preemption-light multiprocessor scheduling of sporadic tasks with high utilisation bound. In: Proc. of the IEEE Real-Time Systems Symposium, pp 447–456

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

Borg A, Wellings A, Gill C, Cytron R (2006) Real-time memory management: Life and times. In: Proc. of the Euromicro Conference on Real-Time Systems, pp 237–250

Brandenburg B, Anderson J (2009a) Joint opportunities of real-time Linux and real-time systems research. In: Proc. of the Real-Time Linux Workshop

Brandenburg B, Anderson J (2009b) On the implementation of global real-time schedulers. In: Proc. of the IEEE Real-Time Systems Symposium, pp 214–224

Brandenburg B, Anderson J (2009c) Reader-writer synchronization for shared-memory multiprocessor real-time systems. In: Proc. of the Euromicro Conference on Real-Time Systems, pp 184–193

Brandenburg B, Calandrino J, Anderson J (2008) 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

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

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

Childs S, Ingram D (2001) 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

Dhall SK, Liu CL (1978) On a real-time scheduling problem. Operations Research 26:127–140

Eswaran A, Rajkumar R (2005) Energy-aware memory firewalling for QoS-sensitive applications. In: Proc. of the Euromicro Conference on Real-Time Systems, pp 11–20

Faggioli D, Trimarchi M, Checconi F (2009a) An implementation of the Earliest Deadline First algorithm in Linux. In: Proc. of the ACM symposium on Applied Computing, pp 1984–1989

Faggioli D, Trimarchi M, Checconi F, Scordino C (2009b) An EDF scheduling class for the Linux kernel. In: Proc. of the Real-Time Linux Workshop

Gosling J, Bollella G (2000) The Real-Time Specification for Java. Addison-Wesley Longman Publishing Co., Inc.

Hand S (1999) Self-paging in the Nemesis operating system. In: Proc. of the USENIX Symposium on Operating Systems Design and Implementation, pp 73–86

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

Kato S, Yamasaki N, Ishikawa Y (2009) Semi-partitioned scheduling of sporadic task systems on multiprocessors. In: Proc. of the Euromicro Conference on Real-Time Systems, pp 249–258

Laplante P (1993) Real-time systems design and analysis. IEEE Press

Lin C, Brandt S (2005) Improving soft real-time performance through better slack reclaiming. In: Proc. of the IEEE Real-Time Systems Symposium, pp 410–421

Liu CL, Layland JW (1973) Scheduling algorithms for multiprogramming in a hard real-time environment. Journal of the ACM 20:46–61

Lopez J, Diaz J, Garcia D (2004) Utilization bounds for EDF scheduling on real-time multiprocessor systems. Real-Time Systems 28:39–69

Marzario L, Lipari G, Balbastre P, Crespo A (2004) 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

Nakajima T, Tezuka H (1997) Virtual memory management for interactive continuous media applications. In: Proc. of the IEEE Interntional Conference on Multimedia Computing and SYstems, pp 415–423

Oikawa S, Rajkumar R (1999) 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

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

Pizlo F, Vitek J (2006) Memory management for real-time java: State of the art. In: Proc. of the IEEE Symposium on Object Oriented Real-Time Distributed Computing, pp 248–254

Rajkumar R, Lee C, Lehoczky J, Siewiorek D (1997) A resource allocation model for QoS management. In: Proc. of the IEEE Real-Time Systems Symposium, pp 298–307

Robertz S (2002) Applying priorities to memory allocation. In: Proc. of the ACM International Symposium on Memory Management, pp 108–118

Srinivasan B, Pather S, Hill R, Ansari F, Niehaus D (1998) 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

Tokuda H, Nakajima T, Rao P (1990) Real-Time Mach: Towards a predictable real-time system. In: Proc. of the USENIX Mach Symposium, pp 73–82

Wang Y, Lin K (1999) 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

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

Yang T, Liu T, Berger E, Kaplan S, Moss JB (2008) 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

Yodaiken V (1999) The rtlinux manifesto. In: Proc. of the Linux Expo