Efficient Soft Real-Time Processing in an Integrated System

Technical Report (UCSC-CRL-04-11)

Caixue Lin and Scott A. Brandt
Computer Science Department
University of California
Santa Cruz, CA, USA
{lcx,sbrandt}@cs.ucsc.edu

Abstract

The rapidly increasing user demands on more powerful computing platforms and application capabilities requires modern operating systems capable of scheduling multiple classes of processes in an integrated way, in which real-time applications are guaranteed to meet their time constraints by using worst case resource reservation and non-critical applications (such as soft real-time) execute in degraded performance by using average case reservation. However these reservation-based resource allocation mechanisms may overbook resources so that the system may waste large amounts of resources in such a way that the slack time is not efficiently consumed. Also, overhead caused by best-effort processes is usually not considered. In this paper, we present for such an integrated system a flexible and efficient resource management mechanism for soft real-time processes in such a way that the slack time is better utilized and a single monolithic server for all best-effort processes to reduce the dynamic scheduling overhead. The simulation results show that our flexible slack time management mechanism for soft real-time processes can result in significantly performance improvement in terms of reducing the deadline miss ratio and tardiness.

Keywords: Integrated Scheduling, Soft Real-Time, Slack Time Management

1 Introduction

The rapidly increasing user demands on more powerful computing platforms and application capabilities requires modern operating systems to support integrated scheduling of multiple classes of processes. The typical example work are the hierarchical scheduler developed by Regehr *et al.* [10] and the flat integrated scheduler RBED by Brandt *et al.* [3]. In such an integrated system, worst case resource reservation or execution is usually used to guarantee the performance of critical real-time applications, such as external event/signal sampling and processing; non critical applications, including soft real-time (*e.g.*, desktop audio/video) and best effort (*e.g.*, compiler), receive the leftover resources and run in degraded performance. However the constant worst case reservation mechanism always overbooks the resources that an real-time application does not really need. That is the system may waste large amounts of resources in such a way that none of the soft real-time and best effort processes can efficiently use the slack time left by the hard real-time processes. Furthermore, in such a system, the overhead incurred by the frequent entrance and exit of a large number of best-effort processes is another important issue since the behavior of best-effort processes is not as predictable as that of soft or hard real-time processes.

In this paper, we present a flexible resource management mechanism in which resources are better utilized to improve the performance of soft real-time and best-effort applications and still guarantee the worst case performance of real-time applications in such a way that no deadline is missed.

First, though both aiming at integrated scheduling, specifically our working system uses constant reservation-based resource allocation mechanism, which is some how different from the dynamic rate-based resource allocation mechanism used in our previous work RBED [3]. With our resource allocation mechanism, hard real-time and soft real-time processes are guaranteed a minimum resource allocation in a fixed interval of time (usually period) and best-effort processes are never starved. However RBED changes allocated resources and application periods dynamically as needed without violating EDF constraints, guaranteeing that hard real-time processes never miss their assigned deadlines and soft real-time processes run in degraded performance specifically depending on their demanding load.

Second, our work focuses on enhancing the system to let the soft real-time processes use the slack time left by the hard real-time processes and other soft real-time processes. That is the slack time is consumed as early as possible by the soft real-time processes which need more resources in the sense that their actual demanded resources are beyond their allocated resources. Though short-term fairness is not guaranteed in such a mechanism, long term-fairness, lower deadline miss ratio and smaller tardiness are achieved.

Third, all best-effort processes with same timeliness properties are considered as one single soft real-time processing server, which schedules each best-effort process in turn.

Our preliminary results show that our flexible scheduling mechanism can result in significantly performance improvement for soft real-time processes in terms of reducing the deadline miss ratio and tardiness (by an average of over 20% in our experiments) compared to the well-known CBS [1] mechanism and the BEBS algorithm [2].

2 Related work

Traditional slack stealing algorithm [7] for priority-driven (including deadline-driven) or fixed-priority system, schedules aperiodic jobs whenever the peroidoc tasks and sporadic jobs have slack, that is, their execution time can be safely postponed without causing them to miss their deadlines. However its main drawback is the incurred huge overhead to do slack computations at high frequency: the extra work that the algorithm (both static and dynamic) must do to determine the amount of the available slack at each scheduling decision time (The computation and thus the overhead always exist whenever there is actually slack or not).

CBS [1] is designed to provide CPU bandwidth reservations to continuous media applications. It provides greedy slack time reclamation mechanism by immediately releasing the next job of an expired process (*i.e.*, it has consumed the reserved resources during its execution) with the deadline set to the end of its next period. Note that this slack time technique can be efficiently applied to acyclic processes (such as CPU bound best-effort processes) which allow to rephrase their jobs' release times (if there are). Here, to rephrase a job's release time means to dynamically adjust the pre-defined old release time of the job to a new one so that the job can be released earlier or later than the old one. However the main drawback of CBS when serving acyclic processes is that it does not guarantee for them a minimum execution time in a fixed interval of time (refer to [8] for more detail). Furthermore, for cyclic processes whose jobs' release times can't be rephrased (such as periodic soft real-time) CBS really does not provide any way to manage slack time.

GRUB [6] is a CBS-alike algorithm. It tends to dynamically allocate excess capacity to the current running server or a few active servers in direct proportion to their processor shares. It has to frequently decide when and the duration during which the slack time (idle time) is constantly available. Furthermore the available slack time are dynamically re-allocated to the needy servers by updating their reservations. These dynamic operations may incur in large overhead. In our case, the scheduler does no dynamic resource re-allocation at all. It knows exactly when and how much the slack is generated and allows it to be used (without incurred in overhead) by those processes which need it most and most urgently (earliest deadline).

The IRIS principle [8] is based on the CBS algorithm. It enhances CBS with a fairer slack reclaiming strategy and guarantees a minimum execution to a task in a fixed interval of time. However this technique can be applied to best-effort applications as that in CBS, but not efficiently applied to periodic or sporadic soft real-time applications whose release times can not controlled or rephrased by the system. Similar to IRIS, BEBS [2] models best-effort applications as aperiodic tasks. Again, it has all the disadvantages as in IRIS. In both IRIS and BEBS, slack time is pushed back to the end of a point only at which it is available to be used fairly. Also in both algorithms, each best-effort process is considered as an independent server. As a result, large overhead may be incurred by the frequent dynamic changes in the scheduling parameters upon every re-schedule and the frequent changes in the numbers of the runnable best-effort processes.

Our previous work RBED [3] provides an CBS-alike slack time management mechanism in which only best-effort processes are allowed to consume the slack left by hard real-time and soft real-time processes. It does so by immediately releasing the next job of an expired best-effort process with the deadline set to the end of its next pseudo-period. Note that RBED assigns dynamic pseudo periods to best-effort processes to enforce EDF scheduling algorithm. Again, with this technique, though the slack time is evenly distributed among the runnable best-effort processes, a new-entered best-effort may cause others to starve; Also it does not allow other classes of processes (notably soft real-time) to take advantage of dynamic slack.

RBED also provides dynamic rate adjustment (similar to the elastic model [4] and VRE model [5]) during schedule in case that extra resources (slack) are available or process demand changes. However frequent dynamic change in scheduling parameters introduces high overhead, which is again not good for an integrated real-time system

3 System Model and Problem Description

3.1 System Model

The system we are considering is an integrated system (as that described by our previous work RBED [3]) consisting of hard real-time, soft real-time and best-effort processes. For simplicity, here the real-time processes (including hard and soft real-time processes) we are dealing with are periodic process, which consists of a sequence of periodically released jobs whose deadlines equal their release times plus the process period. Note that the release time of a periodic process (such as external data sampling and multimedia applications) can not be rephrased dynamically during scheduling.

Our system uses separated rate-based resource allocation and EDF scheduling algorithm (refer to RBED [3] for details) for the processes in the system.

3.1.1 Rate-based resource allocation

The rate-based resource allocation mechanism, which is similar to the reservation-based resource allocation (such as Processor Capacity Reserves [9] and CBS) allocates a fixed minimum resource (here it is CPU execution time) budget to each process in a fixed interval.

In our system, each periodic real-time process T_i consists of sequential jobs $J_{i,k}$, where each job has a release time $r_{i,k}$, fixed period p_i, deadline $d_{i,k}$, and fixed minimum execution budget e_i. So the resource rate (utilization) of T_i is $u_i = \frac{e_i}{p_i}$. All best-efforts processes are associated with a single pseudo periodic server (called BEServer) which has a pseudo period $p_{beserver}$ and a pseudo execution budget $e_{beserver}$ (The detailed management conducted by the BEServer is explained in section 4.2). The total global system utilization is $U_g = U_{HRT} + U_{SRT} + u_{beserver}$, where $U_{\mathbf{X}} = \sum_{T \in \mathbf{X}} T.u$ and $\mathbf{X}$ is the set of all tasks of type X.

Specifically, upon request at process creation time, the rate-based resource allocation mechanism determines the execution budget e_i for process T_i according to the followings rules subject to $U_g \leq 1$:

1. If T_i is hard real-time process, e_i is assigned to its worst case execution time so that all its deadlines are guaranteed to be met. T_i is rejected if $U_g > 1$ upon the allocation.

2. If T_i is soft real-time process, e_i is assigned to an allocated bandwidth, which is equal or less than the worst case. Note that e_i could be any value between its average case execution time and worst case execution time depending on the tradeoff between the soft real-time performance and throughput. T_i is rejected if $U_g > 1$ upon the allocation.

3. For the BEServer, a minimum resource utilization β is reserved for it to guarantee no best-effort processes is starved, *i.e.*, $u_{beserver} = max(\beta, 1 - U_{\mathbf{HRT}} - U_{\mathbf{SRT}})$. $e_{beserver}$ is computed as $e_{beserver} = u_{beserver} \times p_{beserver}$ where $p_{beserver}$ is usually set to a value in such a way that tolerable response time is guaranteed for interactive best-effort processes (I/O bound processes).

3.1.2 EDF scheduling and one-shot mechanism

Our proposed mechanism uses EDF algorithm to schedule processes with resource reservation and overrun protection enforced by a high-precision one-shot timer. Since U_g is never bigger than 1, the EDF scheduling algorithm always provides the required guarantees to the different classes of processes in the system:

1. Every hard real-time process in the system is guaranteed to receive a fixed minimum amount of CPU budget in any of its periods so that it meet all of its deadlines.

2. Every soft real-time process in the system is guaranteed to receive a fixed minimum amount of CPU budget in any of its periods so that it is guaranteed an average performance though some jobs of the soft real-time processes may miss their deadlines.

3. All best-effort are always guaranteed to share the reserved resources for them so that none of them is starved.

Note that process overrun (*i.e.*, allocated budget is consumed before a job finishes) is protected by issuing a one-shot timer to the current running process with the timer counter set to the process' left budget c_i (Note that c_i of a process is initialized to its assigned budget e_i and c_i is then decreasing when the process is running on CPU). Processes are rescheduled when the one-shot timer expires. Here, we only consider soft real-time process overrun since it uses average execution reservation. We assume job is never dropped in such a way that an overrun soft real-time process may still re-start the overrun part of a job upon its next release (Note that an alternative is to notify the application to drop its overrun job if drop is allowed).

An example of process overrun is given in Figure 1, which shows the CPU allocation for a system running three soft real-time processes ($P1$, $P2$ and $P3$) with following configurations respectively: ($p_1 = 6, e_1 = 1.5$), ($p_2 = 8, e_2 = 4$) and ($p_3 = 10, e_3 = 2.5$). Assume the first job of $P1$ has actual execution time 2, which is larger than its budget $e_1 = 1.5$. Thus its first job overruns and the overrun part is pushed back until the next release time. In this case, though the first job misses deadline, the second job (consisting of the overrun part of the first job and the original second job) may still finish before its deadline at time 12 if the original second job has far less actual execution time than the process' budget.

3.2 Problem Description

There are main drawbacks when use the simple EDF scheduling algorithm with the one-shot mechanism for an integrated system in which hard real-time and soft real-time processes may generate slack time at any time because of highly varying execution times but constant reservation. Figure 2 shows the problems when no slack time management technique is used for the EDF scheduler with one-shot

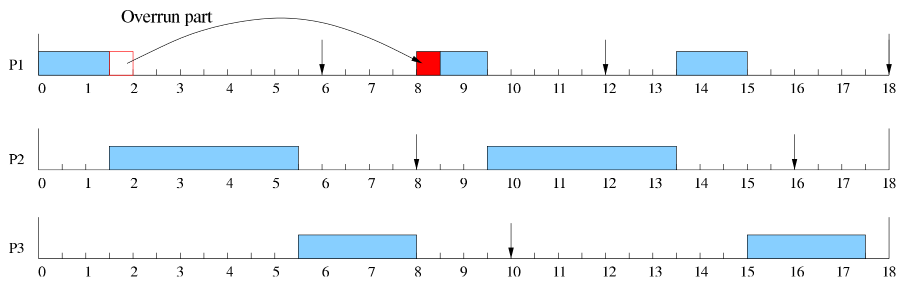
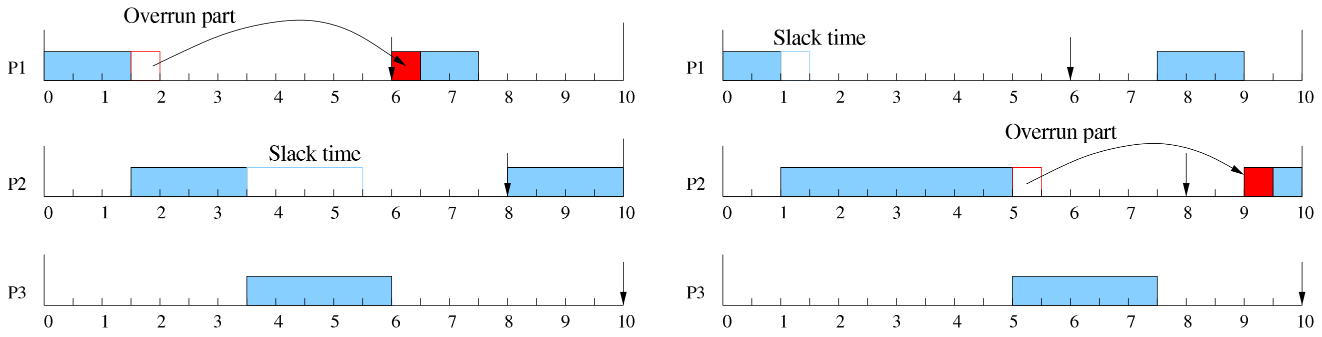


Figure 1. Soft real-time process overrun example



(a) case 1

(b) case 2

Figure 2. One-shot EDF scheduler drawbacks

mechanism. The processes showed in Figure 2 have the same parameters as those showed in Figure 1.

The first drawback is slack time may be not used by a past overrun job of a process. In Figure 2(a), the first job of $P1$ has an overrun part of 0.5; The first job of $P2$ has actual execution time of 2 and thus it generates slack time of 2. Since there is no slack time management employed, the overrun job of $P1$ can't start execution until time 6 and thus misses its deadline.

The second drawback is slack time may be not used by a future (following) overrun job of a process. In Figure 2(b), the first jobs of $P1$, $P2$ and $P3$ finish before their deadlines as expected. However now the second job of $P1$ has actual execution time of 1 and thus it generates slack time of 0.5; the second job of $P2$ has an overrun part of 0.5. Again without slack time management, the generated slack time is pushed back and available until idle point 17. As a results, the overrun job of $P2$ misses its deadline.

4 Design

4.1 Efficient slack time management

The basic principle for our slack time management is to use slack time as early as possible for the processes which need it most. Note that our work focuses on improving the system performance of scheduling soft real-time processes in terms of tardiness and deadline miss ratio, but not fairness in using slack time.

As described in section 3, the currently running process is associated with a one-shot timer which controls the maximum duration of the execution based on the left budget. The one-shot timer starts or updates its counter value when the process starts to run. When the one-shot timer expires, the interrupt handler calls a predefined timer handler, in which the status of the currently running process is set to *expired* and re-schedule is triggered. Note that the overhead on starting the one-shot timer, executing the interrupt handler, executing the timer handler and context switch have to be taken into account for setting the exact counter of the timer so that no extra resource is taken. In re-schedule, the currently expired process is preempted and at the same time next process with the earliest deadline in the run queue is

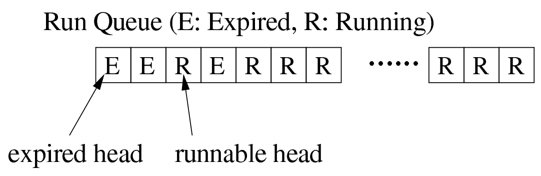


Figure 3. Run queue structure

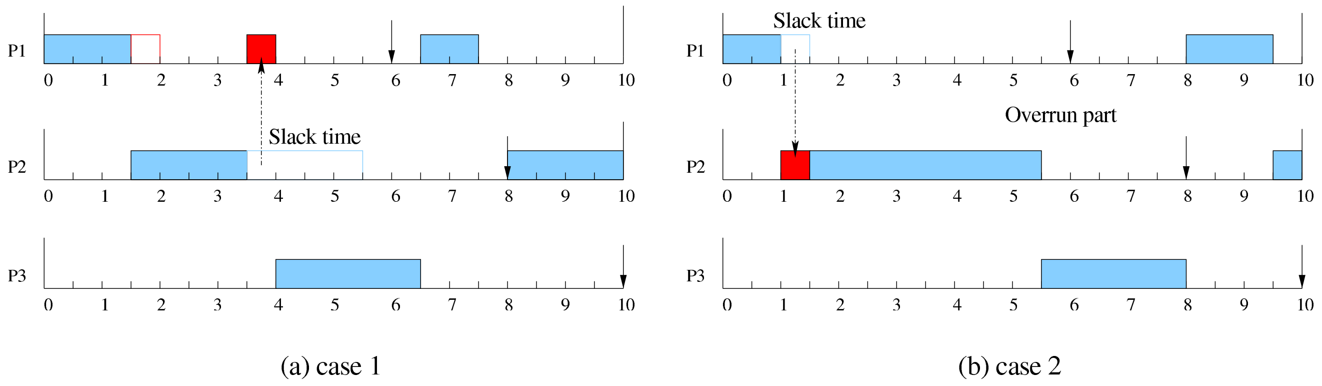


Figure 4. Slack time management for soft real-time processes

picked to run.

When the currently running process completes or blocks before the one- shot timer expires, slack time is generated. Upon process exit (completion) or blocking, re-schedule is again triggered. In this case, the expired process (if there is) with the earliest deadline is selected to run with the old still running one-shot timer (see Figure 4(a) for example). This is done by resetting the status of the selected expired process to RUNNING and artificially creating for it a virtual deadline equal (equivalent) to the deadline of the currently running process which generates slack time. Note that no dynamic rate adjustment is needed during the process. In real-implementation, for reducing overhead purpose, this extra operation can be eliminated by artificially creating a pseudo runnable process which keeps all the live parameters of the currently running process which generates slack time until the one-shot timer expires. In this way, the slack time is allowed to be consumed by the expired processes using EDF algorithm until the timer expires. In case that there is no expired process available, the runnable soft real-time process (the BEServer is an exception, which is detailed in subsection 4.2) with the earliest deadline is instead selected to run. In this way, the slack time is transferred to the future processes until the timer expires.

The run queue is sorted by deadline in such a way that the first job in the queue has the earliest deadline (Figure 3). We use two list heads, expired-head and runnable-head to respectively index the first expired process with the earliest deadline and the first runnable process with the earliest deadline in the run queue. Thus no overhead is incurred to pick a process (either expired or runnable) with the earliest deadline to run.

Our slack time solution solves the problems addressed in Figure 2. The corresponding resulting schedules are showed in Figure 4. Clearly, in both examples, with slack time management, no job misses their deadlines even though there is overrun job.

The advantages of using EDF scheduler with our one-shot timer mechanism support are summarized as follows:

1. Slack time is always consumed as early as possible by processes which need it most. In this way, the tardiness and deadline miss ratio of soft real-time processes are reduced. Note that short-term fairness is not our performance metric.

2. EDF is used as a universal scheduling algorithm both in normal process scheduling and slack time management.

3. Scheduling overhead is low since no dynamic complex computation on scheduling parameters are required. Simulation results show no extra context switch is incurred in our system enforced with new slack time management.

4.2 The BEServer

In our system, in order to reduce the overhead caused by dynamic schedule, all best-effort processes are in turn se-

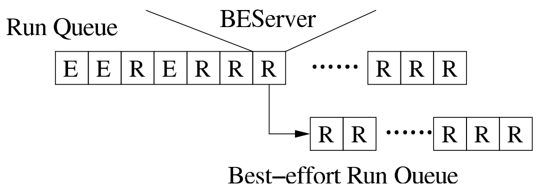


Figure 5. BEServer and best-effort run queue

lected by a BEServer, which is a pseudo soft real-time process scheduled by the EDF algorithm. All runnable best-effort processes are managed by a separate best-effort run queue (Figure 5). The BEServer is runnable if and only if the best-effort run queue is not empty; otherwise it is blocked (suspended). The BEServer is almost the same as a normal soft real-time process, with the exception that BEServer is not allowed to use slack time produced by other real-time processes until there is no expired or runnable real-time process in the system run queue. Thus the soft real-time processes are guaranteed to receive the slack time before any best-effort processes because they have more strict time constraints than those of best-effort processes.

Usually, best-effort processes are divided into two categories: I/O bound and CPU bound processes. In order to provide different timeliness service for them, in stead of using a single best-effort run queue, we can use unix-alike two-level feedback run queues, of which the first one stores the new-entered best-effort processes and the second one stores the scheduled best-effort processes. When the BEServer is scheduled, it always picks the first BE (if there is) in the first queue to run and at the same time the first BEs in the second queue will be removed and added to the end of the first queue.

There are three advantages for using BEServer to manage the best-effort processes. First, since BEServer is a pseudo soft real-time process, other soft real-time processes also can automatically consume the slack time generated by the BEServer when the best-effort run queue is empty. Second, the allocated resource utilization (rate) for the BEServer is not necessarily to be dynamically changed until the demanded workload of real-time processes changes. That is the parameters of the BEServer only depends on the number or load of real-time processes, but not the number of best-effort processes. Thus we eliminate the overhead that would be introduced by the rapidly change in the number of best-effort processes as that in bandwidth preserving or server-based systems, such as CBS and BEBS. Third, A best-effort process at least receives the BEServer budget.

Figure 6 gives the pseudo code for the EDF scheduling algorithm with one-shot mechanism.

5 Preliminary Results

We simulated our slack time management technique and compared its performance to CBS as well as BEBS. For simplicity, we call our one-shot EDF as RBED using EDF.

In the simulations, we ran 5 hard real-time tasks (HRT) and some soft real-time tasks (SRT) at the same time. The 5 hard real-time tasks always take 50% of the CPU usage; The soft real-time tasks increase their usage from 40% to 50% during the experiments; All the soft real-time tasks have the same usage, but may have different periods. Figure 7, Figure 8 and Figure 9 show the performance results in terms of tardiness (fraction of period) and deadline miss ratio respectively with 1 SRT, 2 SRT (using same periods), and 3 SRT (using different periods).

Clearly, from Figure 7, Figure 8 and Figure 9, the average performance in terms of tardiness and deadline miss ratio of RBED (using EDF policy) outperforms CBS and BEBS. However, as showed in Figure 8, BEBS allocates slack time to SRT more fairly than RBED. But once the performance in terms of tardiness and deadline miss ratio is guaranteed to be better, the fairness is not important at all.

Furthermore, we can use other policies, such as randomly pick a process, instead of EDF to pick a SRT to use the slack time. Figure 10 show the performance comparison between RBED (EDF) and RBED (Random). The result turns out that the random algorithm does not work well.

Finally, we also recorded the context switch number for each run in out simulations. The result is by using our slack time management no extra overhead is introduced compared to CBS or BEBS.

6 Conclusion and future work

We present an efficient resource management for soft real-time processes in an intergrated system. This technique employs a simple one-shot timer based EDF algorithm, which allows soft real-time processes to use generated slack time as early as possible. Our simulation results show significant performance improvement for soft real-time processes in terms of smaller tardiness and lower deadline miss ratio by using the slack time management technique other than CBS or BEBS. Also there was no extra

```
EDF_Schedule(){
 //If needed, preempt the currently running process CURR
 if (CURR is a BE process) status = BEServer.status;
 else status = CURR.status;
 switch(CURR.status){
     case EXIT:
     case BLOCKED:
         //make sure we have enough slack time generated
         if (timer.counter < time(context_switch)){
             have_slack = 0; goto done;
         }
         NEXT = expired-head; //the first expired process
         NEXT.status = RUNNING;
         vd = CURR.d; //set the virtual deadline
         have_slack = 1; break;
     case EXPIRED: have_slack = 0;
     case RUNNING: CURR.c = timer.counter; //update the left-budget for CURR
     case default:
done:
         if (runnable-head!=NULL) NEXT = runnable-head; //the first runnable process
         else NEXT = expired-head; //the first expired process
         timer.counter = NEXT.c; //update the one shot timer counter
 }
 context_switch(CURR, NEXT);
}
```

Figure 6. Pseudo-code for the EDF scheduler with one-shot mechanism

context switch incurred in our simulations.

Our future work is to implement the efficient soft real-time resource management technique (including slack time management and best-effort management) in a real system, such as Linux 2.6, and investigate its performance by performing extensive experiments.

Acknowledgments This research was funded in part by Intel. Thanks to Scott Banachowski for his comments and suggestions.

References

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 19th IEEE Real-Time Systems Symposium (RTSS 1998)*, pages 4–13, Dec. 1998.

[2] S. Banachowski, T. Bisson, and S. A. Brandt. Integrating best-effort scheduling into a real-time system. In *Proceedings of the 25th IEEE Real-Time Systems Symposium (RTSS 2004)*, Dec. 2004. To appear.

[3] S. A. Brandt, S. Banachowski, C. Lin, and T. Bisson. Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In *Proceedings of the 24th IEEE Real-Time Systems Symposium (RTSS 2003)*, pages 396–407, Dec. 2003.

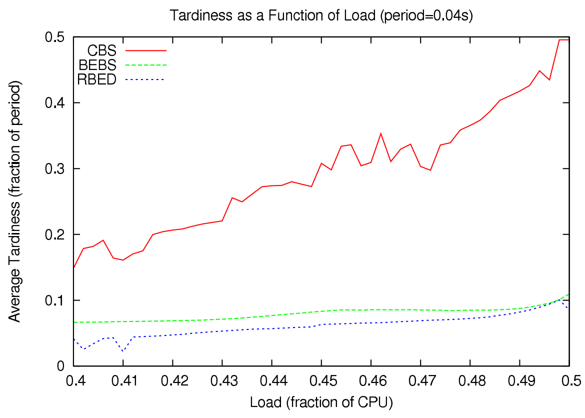
[4] G. C. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Transactions on Computers*, 51(3):289–302, Mar. 2002.

[5] S. Goddard and L. Xu. A variable rate execution model. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, pages 135–143, July 2004.

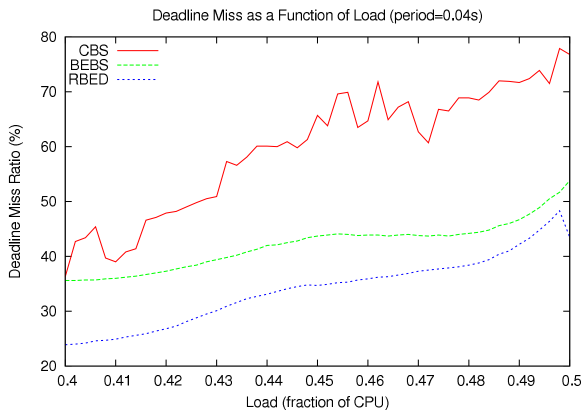
[6] G. Lipari and S. Baruah. Greedy reclamation of unused bandwidth in constant-bandwidth servers. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, pages 193–200, June 2000.

[7] J. W. Liu. *Real-Time Systems*. Prentice–Hall, 2000.

[8] L. Marzario, G. Lipari, P. Balbastre, and A. Crespo. IRIS: A new reclaiming algorithm for server-based real-time systems. In *10th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS04)*, May 2004.

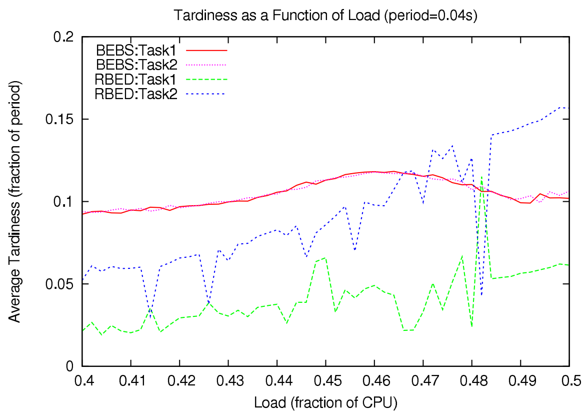


(a) Tardiness

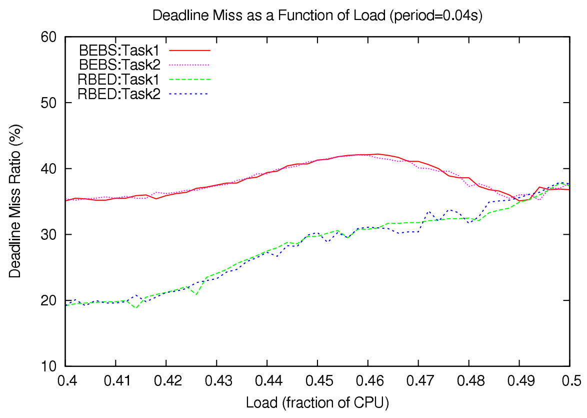


(b) Deadline Miss Ratio

Figure 7. Tardiness and deadline miss ratio in CBS, BEBS and RBED using EDF



(a) Tardiness

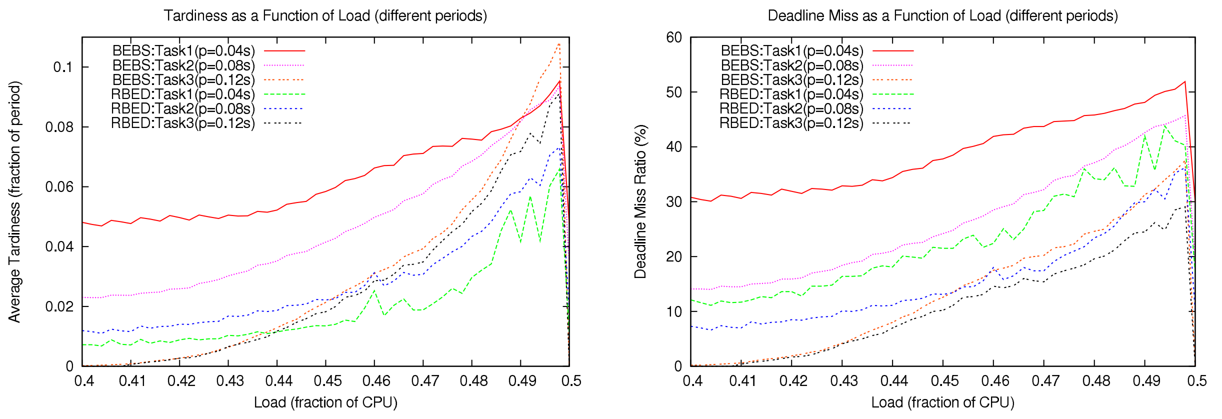


(b) Deadline Miss Ratio

Figure 8. Tardiness and deadline miss ratio in BEBS, and RBED using EDF (two tasks use same parameters, i.e., usage and period)

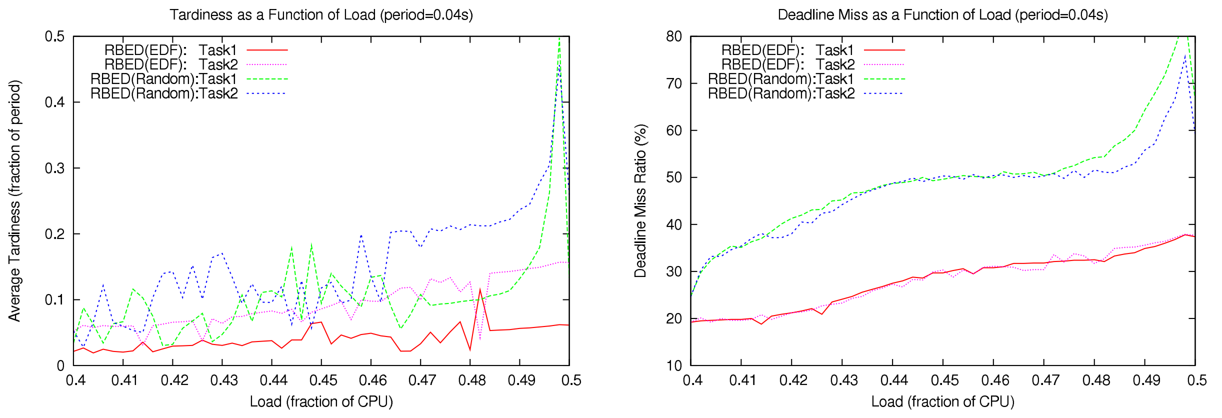
[9] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of the 1994 IEEE International Conference on Multimedia Computing and Systems (ICMCS '94)*, pages 90–99, May 1994.

[10] J. Regehr and J. A. Stankovic. HLS: A framework for composing soft real-time schedulers. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium (RTSS 2001)*, pages 3–14, London, UK, Dec. 2001. IEEE.



(a) Tardiness (b) Deadline Miss Ratio

Figure 9. Tardiness and deadline miss ratio in BEBS, and RBED using EDF (three tasks use same usage, but different periods)



(a) Tardiness (b) Deadline Miss Ratio

Figure 10. Tardiness and deadline miss ratio using EDF and random policy respectively in RBED(two tasks use same parameters, i.e., usage and period)