Queue-based scheduling for soft real time applications

Fabrizio Mulas
University of Cagliari
Cagliari, Italy
Email: fabrizio.mulas@unica.it

Salvatore Carta
University of Cagliari
Cagliari, Italy
Email: salvatore@unica.it

Andrea Acquaviva
Politecnico di Torino
Torino, Italy
Email: andrea.acquaviva@polito.it

***Abstract*—Modern multitasking multimedia streaming applications impose tight timing requirements that demand specific scheduling policies. General purpose operating systems such as Linux (widely diffused even in embedded systems) are not specifically designed for such applications as they must ensure an overall performance level for a wide range of user processes. Realtime versions of general purpose kernels can be used, however since they are designed for hard real-time applications, they impose explicit knowledge of deadlines for all tasks composing the application to set their priorities.**

In this work a novel streaming-oriented scheduling algorithm is proposed, that relies on a standard interprocess communication support for applications composed by multiple pipelined stages communicating by means of message queues. It determines the scheduling order depending on the queue occupancy, for this reason does not require explicit deadline information. It has been developed in Linux OS as a new real time policy, showing that it is relatively easy to integrate in it and, worthily, it does not require modifications of existing applications.

***Keywords*-scheduling; Linux; soft realtime; multimedia streaming.**

I. Introduction

Multimedia applications are increasingly complex and demanding in terms of both computational power and time constraints. A significative example is given by the increasing resolution and frame rate requirements of video streaming applications. When these applications run on top of a general purpose operating system their requirements become very challenging. Indeed, these OSes are currently used in system with demanding networking capabilities, where multiple network flows must be managed. This is true not only for desktop PCs, but also in embedded networking systems such as media gateways, where general purpose OSes are widely used for cost and flexibility reasons. Besides typical network processing, these systems must perform various general purpose processing at line rate such as video decoding, video transcoding, image processing and encryption. In general purpose OSes, the scheduler is not specifically designed for handling real-time requirements even if a standard real-time support does exist in well known general purpose OSes such as Linux or Windows. However, this support is not enough to fulfill the application requirements, basically consisting on giving, to a process defined as "real-time", a static priority higher than any other "conventional process".

Current multimedia applications are composed by a cascade of multiple dependent tasks communicating by means of message queues. For instance, a H.264 decoder is composed by several steps including motion compensation, entropy decoding, dequantization, inverse Discrete Cosine Transform (DCT). Furthermore, multimedia frameworks such as GStreamer create complex multimedia applications by chaining several stages [1]. In both cases, the frame rate (i.e., QoS) requirements are backward propagated from the last stage to the previous ones. A general purpose scheduler, such as the Linux one, is not aware of task dependencies and timing constraints, but only looks at how much a task is demanding in terms of CPU utilization.

The "conventional process" scheduler is designed to promote the so called I/O bounded applications, by giving them a high dynamic priority. These are characterized by small (compared to the timeslice) CPU bursts interleaved to large I/O access periods. CPU bounded ones, instead, are characterized by much larger CPU bursts, and thus are given a smaller dynamic priority. This is because I/O applications are supposed to interact with the user and hence the OS attempts to reduce their latency. On the other side, the real-time process scheduler in Linux implements either a FIFO or a Round-Robin policy. Both of them, as it is going to be shown in this paper, do not take into account actual requirements of tasks, leading to QoS degradation especially in high CPU utilization conditions.

An additional limitation of general purpose OSes arises in presence of multiple real-time applications running simultaneously, as in the context of media gateways, where several streams need to be decoded at the same time to feed multiple network connections. Here the computational power must be allocated to multiple decoding applications having heterogeneous QoS requirements, such that all they perceive a degradation proportional to their QoS requirements. This can be hardly achieved using general purpose OSes that lack the concept of fairness related to the QoS.

Putting it all together, general purpose schedulers are not longer suitable to modern multimedia applications ([2], [3]). Nevertheless, they are still common in Windows family, Linux, and all other variants of Unix such as Solaris, AIX and BSD (see [4] for further details).

An alternative solution would be to adopt hard real-time schedulers, that are specifically developed for scenarios where

 ISBN: 978-1-61208-129-8

deadlines must be strictly respected (e.g., life-safety critical applications). The counterpart is that they are hard to manage and require to explicitly provide the scheduler with timing constraints of applications (i.e., deadlines) that must be hence modified accordingly. There are situations where the requirements about the deadlines are not strict, that is, a certain amount of them can be tolerated (for example, in multimedia streaming): it is the case of soft-realtime applications.

In this paper a variant of the Linux scheduler is proposed, called *queue-based scheduler* (QBS) that deals with soft real-time streaming applications composed by multiple pipelined stages. QBS is inherently aware of QoS requirements of multitask applications similarly to real-time schedulers, but does not require application modifications, as general purpose ones. In order to achieve this goal, it monitors the intertask communication, thus requiring the instrumentation of the communication and synchronization library.

QBS implicitly assumes that applications are composed by multiple pipelined stages that communicate by means of queues of messages. Such applications follow a data-flow paradigm, where tasks continuously process frames arriving in their input queue and produce frames on their output queue for the next processing stage. Figure 1 shows an example of such paradigm (H.263 decoder). Most modern multimedia applications are realized in such a manner (e.g., audio/video decoders). The application output queue is read at fixed time intervals (by a *consumer*) and if it is found empty a deadline miss occurs.

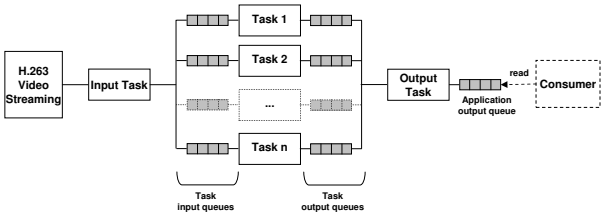

Fig. 1. Pipelined multi-stage application scheme (H.263 Decoder)

The main idea behind QBS is to monitor the queue occupancy level of all queues in the system and to take scheduling decisions based on this information. Basically, QBS seeks the emptiest queue in the system and schedules the process or task writing into it (given that it is in *running* state). Thus QBS can quickly react to situations that may lead to deadline misses, exploiting the feedback from the queues.

In the considered application model, QoS is preserved as long as there are data items available in the application output queue (that is, the last queue of the application) when they are needed by the final consumer stage. This leads to two very important considerations. First, the application output queue can be even empty in some periods of time without necessarily having misses (that is, there is not a miss if the output queue is empty when the consumer does not read data from it). Second, in general intermediate stages have less stringent timing requirements (because they do not generate misses directly) .

The queue feedback approach ensures a more effective CPU time allocation to each task, based on its real and actual QoS requirements. From a practical point of view, the occupancy level of the output queue of a task is used as a measure of its CPU utilization needs. A deep explanation of that, together with a detailed description of the proposed algorithm, is provided in Section III. To test its effectiveness, the scheduler has been implemented inside the Linux OS and the standard System V message queue library has been instrumented to support monitoring features. Thanks to this implementation, various sets of experiments have been carried out, using multiple video decoding applications. Experiments compare the deadline miss rate of QBS w.r.t. both default real-time and conventional process scheduler in case of single and multiple decoding applications having heterogeneous QoS requirements. Results demonstrate that QBS improves the deadline miss rate in high CPU utilization conditions and provides better CPU resource allocation, that is, proportional to frame rate requirements.

The rest of the paper is organized as follows: Section II describes related work in the area of scheduling for real-time and multimedia applications. Section III full details the QBS algorithm, Section IV explains why Linux has been chosen as testbed platform, Section V describes the implementation while Section VI shows the experimental results. Section VII concludes the paper.

II. Related Work

In literature many approaches have been proposed to manage soft real-time applications in commodity OSes. [5] performs a deep evaluation of how clock interrupt frequency influences the response time of multimedia applications. Their study aims at helping tuning existing schedulers. Similarly, other techniques as soft timers [6], firm timers [7] and one-shot timers have been proposed to significantly enhance response time. However, none of them proposes a new scheduler algorithm but rather latency reduction techniques.

On the other side, many real time schedulers have been proposed. SMART [8] is a scheduler for multimedia and real time applications implemented in UNIX-like OSes. It schedules real time tasks even trying to avoid the starvation of conventional processes, but it requires deep modifications of existing applications. In fact, applications have to communicate their deadlines to the scheduler, which can also return feedbacks to enable some proactive countermeasure (e.g., remodulate their workload in order to meet the deadline). On Linux, some examples are Linux/RK [9], RTE-Linux [10], Linux-SRT [11] and RTLinux [12]. These all have the same general drawbacks of real-time schedulers (i.e., programmers must use a dedicated interface to exploit these services). Other approaches explicitly require user intervention to specify the needs (in terms of priority) of the processes or of a class of processes (e.g., multimedia applications) [11] [13].

The algorithm proposed in this paper (QBS) provides QoS sensitive scheduling without requiring explicit user awareness and modification of existing applications, given that they

 ISBN: 978-1-61208-129-8

follow the message queue paradigm. As mentioned in the introduction, this model adheres with the one of modern multimedia applications and frameworks.

III. Queue-based Scheduling Algorithm

The idea behind the proposed algorithm is to exploit the level of the interprocess communication queues as indication of task requirements and consequently to grant CPU time proportionally to that. To better explain that, let us consider a simple example of two applications, A and B, with a CPU need of 65% and 55% respectively (that is, the system is overloaded). Running them in a standard operating system, without any knowledge of application requirements, A and B will receive more or less the same treatment (i.e., about 50% of CPU each), thus A will experience a worse QoS with respect to B. From the point of view of the queues, those of A will be more empty, in average, than those of B. Instead QBS monitors all queues in the system and tries to level them. As a consequence, comparing to the previous case, A will receive more CPU time than B, thus reducing the QoS gap between the two applications (i.e., A will have less deadline misses than before and B a little more than before) and assuring a CPU time sharing proportional to their needs (i.e., both applications will be penalized in a proportional manner rather than in the same way). Furthermore, it is worth noting that QBS, exploiting the feedback from the queues, is able to quick react to situations that potentially lead to deadline misses. For example, if a queue suddenly becomes empty, QBS notices that and properly reacts to fill it.

Algorithm 1 describes how QBS functions. Let Q_n be the nth queue, Q_{Ln} be its level (by definition, Q_L is an integer non-negative number) and let N be the total number of queues in the system, at any moment. Let T_n be the last scheduled time of Q_n's producer. QBS basically finds the most empty queue in the system and schedules the task that writes in it (the producer). Note that in the paradigm used each queue has only one producer and one consumer. If as a result of the search two or more queues are found at the same minimum level, QBS chooses the oldest scheduled producer, that means the process that less recently has been executed in CPU. The $scheduleProducerOf()$ function schedules the producer of the queue passed to it as argument.

Algorithm 1 Queue-based scheduler algorithm

Every decision instant do:

```
1: Q_min = Q_1
2: T_min = T_1
3: for n = 1 to N do
4:     if (Q_Ln < Q_Lmin) OR (Q_Ln = Q_Lmin AND T_n < T_min) then
5:         Q_min = Q_n
6:         T_min = T_n
7:     end if
8: end for
9: scheduleProducerOf(Q_min)
```

The last point to analyse is how frequently QBS should be executed. There is clearly a trade-off here, indeed: choosing a high frequency achieves a better leveling of the queues, but, on the other hand, it increases the number of context switches, thus causing a higher overhead. Thus, it has been chosen to maintain the concept of Linux *timeslice*: every process can consecutively use the CPU till a maximum amount of time (i.e., the timeslice), at the end of which the scheduler is called and the current process (most of the times) is preempted and another one is scheduled.

A. QBS Complexity

The algorithm's complexity is related to the need of scanning all queues in the system to find the most empty one. Thus QBS would have a linear complexity, that is $O(n)$ (where n is the total number of active queues in the system). Given that the scheduler is called very frequently, it is mandatory to reduce its complexity as much as possible. Then it has been reduced to $O(1)$, that means it no longer depends on the number of the queues. This result has been achieved adopting a special data structure to keep trace of all queues and considering that, at any moment in time, the only ones that could change are those read and written by the task currently in execution. So, when the scheduler is invoked, it quickly updates in the structure the information about the only queues that could have been changed. Hence, the time taken for this operation is constant ($O(1)$). The details of how it is implemented are described in Section V

IV. Testbed System Description

QBS has been implemented in Linux 2.6, thanks to its open source nature and widespread diffusion. Indeed it is used in desktop PCs, many server systems (e.g., web, mail, dns, routers, etc.) and, recently, in mobile platforms too. One of the most notable examples of that is probably Android [14], the Google OS for smartphones, based on Linux and widely thought to reach a leading position in the market very soon. QBS aims at be adopted in above systems and even in small/medium multimedia servers (e.g., audio/video on demand, voip, etc.), where expensive high specific solutions (e.g., real time OSes) are not affordable and commodity operating systems are the usual choice. Thus, in all these systems the standard Linux scheduler is adopted. For all these reasons it has been decided to compare QBS versus Linux standard policies. The following section (IV) details these policies.

Linux standard distributions come with three policies (some slight variations are possible depending on kernel versions, but they are basically the same): SCHED_NORMAL, SCHED_RR and SCHED_FIFO. The first one is the default policy for all tasks. It is a relatively complex algorithm that deals with conventional processes (i.e., not real time processes). It continuously attempts to identify interactive applications from CPU intensive ones, using the common mechanism (common to many OSes) described early (in the Introduction): processes that spend most of their time waiting for I/O operations are supposed to be interactive, while those that heavily exploit the CPU fall in the second category.

 ISBN: 978-1-61208-129-8

Then the scheduler grants more priority to the interactive ones, in order to reduce their latency. Unfortunately nowadays interactive multimedia applications are CPU greedy too, thus they are penalized by this mechanism ([2], [3]). For this reason this policy is not adequate for managing modern CPU-demanding interactive applications (this is demonstrated in Section VI-C).

SCHED_RR and SCHED_FIFO are both real time algorithms: basically the former (round robin policy) equally shares the CPU time among tasks, while the latter (fifo policy) grants all CPU time to the first arrived process as far as it uses it, after that it schedules the next task in the FIFO queue. Thus the last one, given its fifo behaviour, is not adequate for multimedia applications (it does not treat all processes fairly). Instead round robin (RR) performs quite well and consequently has been chosen as the main algorithm to confront against (Section VI-C). It must be noted that Linux real time policies are intended to manage soft real time processes. To specify a task as a real time one, the programmer needs only to state that using a system call. No any other modification is needed. Alternatively, the user can set it using the *chrt* linux command, without any modification to the application code.

V. QBS Implementation Details

This section describes how QBS has been implemented in a standard Linux kernel. In particular, all details are referred to kernel 2.6.20.16.

The Linux scheduler picks up the next task to be executed from the top of a specialized task queue. Thus, the main routine of QBS (i.e., the code that implements the algorithm and chooses the next task to be scheduled) is called just right before this choice, in such a manner to put the process selected by QBS on top of that queue. In this way, the standard Linux scheduler will find in it the task chosen by QBS.

In Section III the core algorithm has been described and in Section III-A it has been stated that its complexity is $O(1)$. All above has been accomplished using the structure showed in Figure 2. It is an array of simply linked lists, where MAX represents the maximum possible number of items in a queue (i.e., a System V message queue). Each element of the lists is a *queue identifier*, a special structure that points to an allocated queue. The key point here is that, at any moment in time, each element in the nth list (i.e., that at position n in the array) points to a queue that has n items in it (a that time). Thus the algorithm described in Section III is implemented in this way: it scans the array starting from 0 and selects the first element found. Hence, it points to the most empty queue in the system, as requested by the algorithm. Using this structure, the algorithm needs to scan at maximum MAX array items, resulting in a constant seek time (i.e., $O(1)$ complexity).

The queue identifier is composed by three fields: (i) lid is a pointer to a queue; (ii) $timestamp$ represents the last scheduled time of the producer of that queue; (iii) $next$ is a pointer to the next element in the linked list. It must be noted that in each list, all elements are ordered in a temporal way using the timestamp, from left to right, where on the left there is the oldest one. Hence this assures that the first element found during the scan of the array represents both the producer of the most empty queue in the system and, among all queues at the same level, the oldest scheduled one. This structure assures that the time spent for selecting a task is constant ($O(1)$), because it depends on neither the number of the tasks, nor the number of the queues.

This structure is updated every time the scheduler is called: as a further optimization, it checks only the queues modified by the last executed task and, if needed, moves the corresponding identifiers in the correct array position.

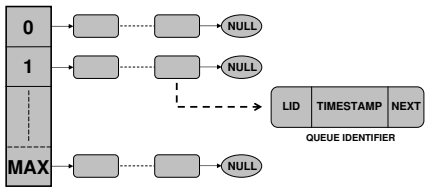

Fig. 2. Array of simply linked lists of queue identifiers

VI. Experiments

This section describes the experimental setup (Section VI-A), the objectives of experiments (Section VI-B) and finally the tests that have been performed (Section VI-C).

A. Experimental Setup

A dedicated machine has been set up for all experiments, equipped with a CPU Athlon XP 1100 GHz and with 512 MB of RAM. For the reasons explained in Section IV, the Linux standard round robin policy (SCHED_RR) is the primary algorithm QBS is compared against. Nevertheless, some comparisons versus the SCHED_NORMAL (conventional) algorithm have been performed too. Several experiments have been set up using many instances of two different applications, both following the message queue paradigm described early in Section I.

The first one, depicted in Figure 3, is composed by synthetic tasks (i.e., they perform some useless work). Each of the first three tasks puts data in its output queue, while Task 4 reads data from all its input queues, performs some elaboration, and puts the result in its output queue.

Instead the second application is a real H.263 decoder, already showed in Figure 1. The movie to be decoded is fully loaded in RAM before the start of experiments, in order to avoid possible bottlenecks reading it from the hard disk. Then the memory is locked to prevent swapping (that could alter the results). All these operations are done by the *Input Task* (see Figure 1), that then decomposes each frame of the video in *n* parts and puts them in the next proper queue. Each following task (*Task 1 to n*) elaborates the nth part of the frame. In the end, the *Output Task* reassembles the decoded frame, performs some elaboration and puts it in the application output queue.

It must be noted that both applications use the System V message queue library. All operations on queues (read and

 ISBN: 978-1-61208-129-8

write) are blocking, that means if a process attempts to read in a empty queue or to write in a full one, it is suspended and automatically woken up as soon as this situation changes.

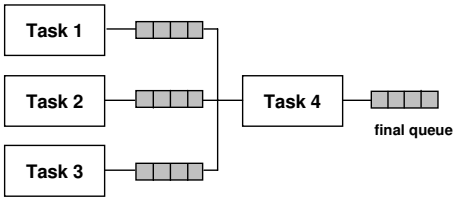

Fig. 3. Synthetic task application

B. Objectives

In the kind of applications that are being considered, an important metric to be taken into account is the quality of service (QoS). Indeed the output queue of each application instance is read at a fixed frequency, depending on the wanted frame rate, and if it is found empty a deadline miss occurs. The fewer the deadline misses, the greater the QoS, thus they should be as few as possible.

As above noted, the total QoS is a significant metric, nevertheless is not the most important one. Indeed, a more valued characteristic is its uniformity, both per and among applications. In order to explain that, consider the case where all instances are perfectly identical: it is not desirable to have a decoder that performs very well while another is working very bad, but rather to have all them with the same QoS level (uniformity among applications), at any time (uniformity per application, i.e., the performance of each application is constant in time). Generally speaking, considering application instances with different requirements, ideally each one should get a QoS proportional to its needs.

The experiments aim at demonstrating that QBS, with respect to standard Linux policies, is able to achieve a better total QoS, a better QoS performance uniformity (both per and among applications) and to provide a QoS proportional to application requirements.

C. Tests and Results

In order to compare the algorithms in real-world situations, a media server has been set up using many instances of the two decoders described before. Thus, many experiments with several instances of such applications running in parallel have been carried out, varying their parameters, as task workload, frame rate, and so on.

1) Synthetic Decoder: Some experiments with the synthetic application have been executed, comparing against both SCHED_RR and SCHED_NORMAL. Figure 4 shows the deadline misses (in percentage with respect to the total number of reads at the application output queue) versus the frame rate, running two application instances in parallel. The miss rate plotted is the average between the two values (note that each application has its own number of deadline misses). In these experiments QBS performs better than the others, having always less misses. Furthermore it sustains a higher frame rate without having QoS worsening (namely, 26.4 fps versus 25.8 fps for SCHED_RR and 22.9 fps for SCHED_NORMAL).

This experiments revealed that SCHED_NORMAL is not adequate for comparing versus QBS: indeed numerical results (not reported in the paper) show that there is a great gap of performance between the two application instances. For example, it can happen that one application has zero misses for a very long time while the other has 15% of it. This is because SCHED_NORMAL is not thought to deal with soft real time processes and furthermore it continuously tries to prioritize interactive tasks (this mechanism is described in Section IV). This is the reason (for fairness) why it has been chosen to not compare against it anymore .

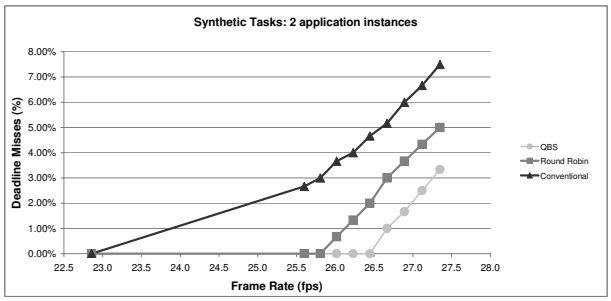

Fig. 4. Synthetic tasks: deadline misses versus frame rate

Using a debug monitoring infrastructure, the behaviour over the time of all queues have been carefully analyzed, observing that QBS is able to level them (in average) while RR shows great differences. It is possible to observe this behaviour in Figure 5. For example (RR case), some queues are totally full while others are completely empty. This suggests the idea that if a queue is always almost empty (in average) and another is in the opposite condition, probably the CPU time could be more fairly distributed (i.e., more CPU time than strictly needed is granted to the task which output queue is fuller). Instead QBS shows the capacity to better level all queues in the system, in average, suggesting a smarter CPU repartition among tasks.

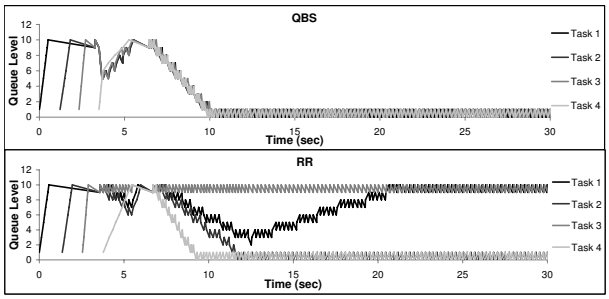

Fig. 5. Queue levels over time

2) H.263 Decoder: In the following examples both algorithms have been much more put under stress, using many instances of the H.263 decoder. Experiments have been carried

 ISBN: 978-1-61208-129-8

out ranging from six parallel instances up to eighteen (note: in this set of experiments all instances are perfectly identical and the input file is the same). Figure 6 shows the deadline misses (in percentage) versus the frame rate for six applications (note: the value is the average among all applications). It is possible to see that QBS performs slightly better (similar results apply for the other above mentioned cases, that is with more than six decoders).

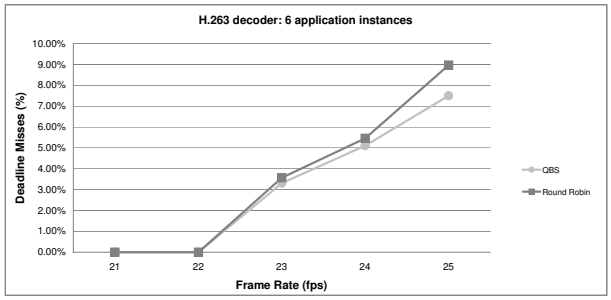

Fig. 6. H.263 decoder: deadline misses versus frame rate

However, the differences are really tiny. But, as it has been explained in Section VI-B, it is more important to assess the QoS uniformity. The two plots in Figure 7 show the miss percentage over the time for each application (eighteen decoders at the same fixed frame rate). Even if it is not possible to distinguish every single application, its aim is to display how QBS is able to much better level the QoS among decoders. Indeed the lines in the QBS plot appear closer each others. To numerically quantify this behaviour, the standard deviation of deadline misses among decoder instances has been calculated, at fixed interval times. The results reveal that standard deviation values in the RR case are roughly three times higher (the average values are 2.0 and 6.1 for QBS and RR, respectively). Thus, RR at any moment in time causes quite big differences among decoders, meaning that some applications are performing much better that others. Another important aspect, not clearly distinguishable from the plots, is that this not uniformity changes also in the time (for RR). That is, given a certain decoder, its QoS oscillates a lot over the time (this is not a desirable behaviour). This happens much less in QBS. Table I numerically points out that, showing the standard deviation of deadline misses (in percentage) of each decoder instance. It is worth noting that has been plotted the case with eighteen decoders, the most stressing one for the algorithm: with less instances QBS performs even better. Carefully observing the Figure 7 in the QBS case, it is possible to see a sort of periodic trend. This is due to three main reasons: (i) the workload varies from frame to frame, depending on their complexity; (ii) all decoders read from the same source file and their application output queue is read at the same instant, hence all tasks have a similar workload at any moment in time (with a certain flexibility due to the queues that intrinsically function as a buffer); (iii) it has been previously stated that for avoiding bottlenecks the video is full loaded in RAM before the starting of experiments, but due to memory space restrictions, a longer duration is simulated re-reading the same movie several times. The first two points explain why all decoders have always similar workloads and their variations over the time, while the last one justifies the periodic trend. To prove that an experiment similar to the above one has been executed, loading only one frame in RAM: RR continues to behave as before (as in Figure 7) while QBS, plotted in Figure 8, now shows a flat trend, without peeks and periodic shapes.

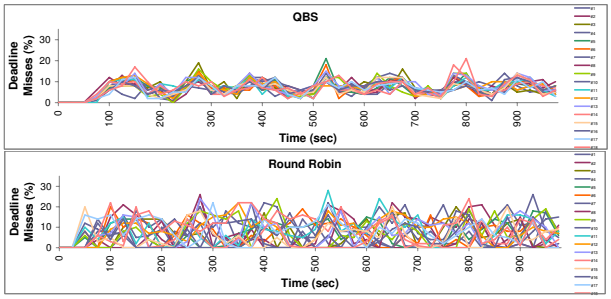

Fig. 7. Eighteen H.263 decoders: deadline misses over time

TABLE I
STANDARD DEVIATION OF EACH H.263 DECODER INSTANCE

	# Decoder Instance																	
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	Average
QBS	3.7	4.0	4.0	3.5	4.1	4.0	3.4	3.2	4.3	3.4	3.7	3.5	4.1	3.7	3.6	3.2	3.4	4.4	3.7
RR	7.1	7.7	6.3	6.2	5.8	6.0	6.5	7.0	6.9	6.0	6.9	6.0	6.9	6.5	6.2	5.9	6.2	7.4	6.5

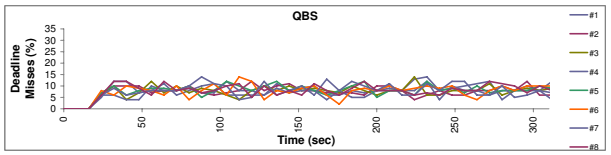

Fig. 8. Eight H.263 decoders: deadline misses over time

Previous experiments have been performed using several instances of the same decoder (either synthetic or real), with the same workload of internal tasks and the same frame rate. They aimed at more easily pointing out some characteristics of both algorithms. In order to assess their behaviour in real scenarios, where applications can have any possible combination of workload and frame rate, other experiments have been carried out, varying these parameters too. Plots in Figure 9 sketch the deadline misses over the time for a case in which there are twelve H.263 decoders at 10 fps and one at 20 fps, for each algorithm. RR causes a higher number of deadline misses in the faster instance (32.8% in total) while none of them in the slower ones (0.0% in total). This is because RR equally shares the CPU time among tasks, without knowledge of their requirements. That means that each decoder, being composed by the same number of tasks, receives the same slice of CPU time. Instead QBS is fairer, indeed observing the queues it recognizes that the faster decoder has a higher CPU need and grants it more CPU time. Hence QBS reduces the gap in QoS between the two application categories (with respect to the previous case), causing less QoS worsening in

 ISBN: 978-1-61208-129-8

one case (10.9% in total) and more in the other one (2.95% in average among decoders).

In order to confirm this positive behaviour of QBS, other experiments have been realized, using eleven identical decoders all at the same frame rate, but with one of them with a much higher workload of its internal tasks (i.e., its tasks perform heavier elaboration). The results (not reported here) are very similar to the previous case, confirming such behavior.

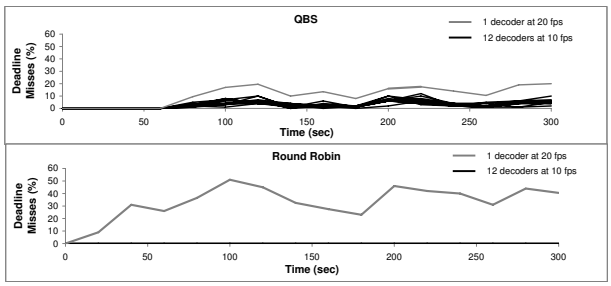

Fig. 9. Thirteen H.263 decoders with different frame rate: deadline misses over time

Finally, one last experiment has been set up, using twelve decoders with incremental workload: the second decoder has a higher workload than the first one, the third one a higher workload than the second one, and so on. Both algorithms show a step results among QoS of applications, as expected, but QBS distributes the performances in a more uniform manner (with respect to RR). Figure 10 plots the results whilst the numerical values are in Table II.

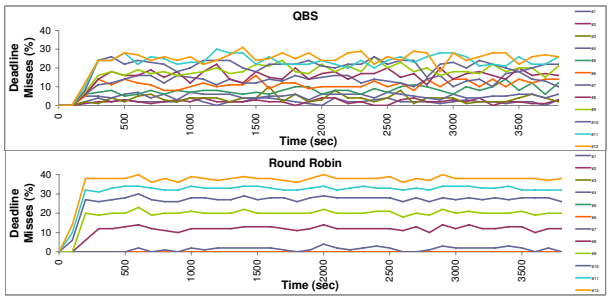

Fig. 10. Twelve H.263 decoders with incremental workload: deadline misses over time

TABLE II
TOTAL DEADLINE MISSES (%) OF EACH H.263 DECODER INSTANCE

	# Decoder Instance											
	1	2	3	4	5	6	7	8	9	10	11	12
QBS	2.1	2.0	3.5	4.9	7.8	10.7	12.8	15.3	17.7	20.5	22.6	24.0
RR	0.0	0.0	0.0	0.0	0.0	0.0	1.4	11.4	19.3	26.3	31.6	36.4

VII. CONCLUSIONS AND FUTURE WORK

Nowadays multimedia applications are widespread in several fields and there are many situations where they are executed in commodity operating systems, such as devices for playing audio/video or small/medium voip servers. General purpose OSes do not provide adequate support to them. The proposed scheduling algorithm (QBS) outperformed standard Linux policies, both in QoS and uniformity performance among application instances. QBS has been validated against various utilization scenarios, using both real and synthetic multimedia applications. Finally, it is relatively easy to integrate in a standard distribution and does not require any modification of existing applications.

We are working to further improve it in several ways, for instance experimenting priorities among queues. We also plan to extend it for multiprocessor systems.

REFERENCES

[1] GStreamer, "Gstreamer multimedia framework." [Online]. Available: http://www.gstreamer.net/

[2] J. Nieh, J. G. Hanko, J. D. Northcutt, and G. A. Wall, "Svr4unix scheduler unacceptable for multimedia applications," 1993.

[3] Y. Etsion, D. Tsafrir, and D. Feitelson, "Desktop scheduling: how can we know what the user wants?" in *NOSSDAV '04: Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*. New York, NY, USA: ACM, 2004, pp. 110–115.

[4] Y. Etsion, D. Tsafrir, and D. G. Feitelson, "Human-centered scheduling of interactive and multimedia applications on a loaded desktop," ,, Tech. Rep., 2003.

[5] Y. Etsion, D. Tsafrir, and D. Feitelson, "Effects of clock resolution on the scheduling of interactive and soft real-time processes," *SIGMETRICS Perform. Eval. Rev.*, vol. 31, no. 1, pp. 172–183, 2003.

[6] M. Aron and P. Druschel, "Soft timers: efficient microsecond software timer support for network processing," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 197–228, 2000.

[7] A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole, "Supporting time-sensitive applications on a commodity os," in *OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*. New York, NY, USA: ACM, 2002, pp. 165–180.

[8] J. Nieh and M. S. Lam, "The design, implementation and evaluation of smart: a scheduler for multimedia applications," in *SOSP '97: Proceedings of the sixteenth ACM symposium on Operating systems principles*. New York, NY, USA: ACM, 1997, pp. 184–197.

[9] S. Oikawa and R. Rajkumar, "Linux/rk: A portable resource kernel in linux," in *In 19th IEEE Real-Time Systems Sumposium*, 1998.

[10] Y.-C. Wang and K.-J. Lin, "Enhancing the real-time capability of the linux kernel," in *Real-Time Computing Systems and Applications, 1998. Proceedings. Fifth International Conference on*, Oct 1998, pp. 11–20.

[11] S. Childs and D. Ingram, "The linux-srt integrated multimedia operating system: Bringing qos to the desktop," in *RTAS '01: Proceedings of the Seventh Real-Time Technology and Applications Symposium (RTAS '01)*. Washington, DC, USA: IEEE Computer Society, 2001, p. 135.

[12] M. Barabanov and V. Yodaiken, "Real-time linux," *Linux Journal*, 1996.

[13] M. A. Rau and E. Smirni, "Adaptive cpu scheduling policies for mixed multimedia and best-effort workloads," in *MASCOTS '99: Proceedings of the 7th International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*. Washington, DC, USA: IEEE Computer Society, 1999, p. 252.

[14] Google, "Android operating system." [Online]. Available: http://www.android.com/

 ISBN: 978-1-61208-129-8