QoS support in the X11 Window Systems

Nicola Manica, Luca Abeni, Luigi Palopoli

University of Trento

Trento - Italy

nicola.manica@gmail.com, luca.abeni@unitn.it, palopoli@dit.unitn.it

Abstract

In this paper, we consider the problem of providing QoS guarantees to the execution of applications using the X11 window system. In particular, we offer a system level analysis of the issues encountered when using X11 to serve real-time applications. By using a tracer developed for the purpose we analyse in depth the internal behaviour of the system. The result of the analysis puts on display the adverse effect played by a non real-time scheduler on the performance of time-sensitive applications. Based on this analysis, we propose an alternative solution based on the CBS scheduler and prove its effectiveness by an extensive set of experiments on real hardware.

1 Introduction

In the past few years, we have observed a clearly established trend toward the use of computer based devices for multimedia applications. The growing commercial fortune of such networked applications as IPTV, YouTube and video servers is a clear indicator of a paradigm shift in the way most people use their personal computers and, generally speaking, their computer based devices. The gain of using computers for this class of applications is evident in terms of flexibility and cost effectiveness. A computer can be used to run multiple and heterogeneous applications at once. Moreover, it is easy to upgrade the software to support new multimedia compression standards and sophisticated sound technologies, as soon as they become available.

This huge potential though poses formidable challenges to the designers of Operating Systems and of network protocols. Indeed, a multimedia application is inherently time-sensitive: uncontrolled fluctuations in latency and framerate defy the patience of any user who expects to watch TV or talk to VoIP phone with a Quality of Service (QoS) comparable to the one experienced with traditional dedicated hardware solutions. To this regard, resource sharing is known to play an adverse role since it introduces scheduling delays that are not easily predictable when designing the application (since they depend on the workload of the system). In contrast, what we need is that a time-sensitive application receives a dedicated fraction of a shared resource in time, regardless of the behaviour of other applications. This property is called *temporal protection* [4] and it is the natural complement of memory protection, which prevents interference in the concurrent access of a set of applications to a limited memory. For years, researchers have been confronted with the challenging effort of designing scheduling algorithms that feature temporal protection. A first important class of algorithms designed to this purpose approximates the Generalised Processor Sharing concept of a *fluid flow* allocation, in which each application using the resource marks a progress proportional to its weight (for example, see some Proportional Share algorithms [21]). Similar are the underlying principles of a family of algorithms known as Resource Reservations [13, 15, 1, 16], which associate to each application a pair (Q, P) guaranteeing that it receives at least Q units of time every P.

In this context a relatively marginal importance has been attached to the problem of building a window system for real-time applications. This lack of attention is not, in our opinion, well deserved. Indeed, for instance, it is of little use to have a very fine grained allocation of the CPU time for a MPEG player if the projection of the movie into the window can be stalled by a non real-time application scrolling a huge amount of textual data in a different window. On the other hand, designing a real-time window system (RTWS) is surely to be considered a challenging activity because a difficult balance has to be found between contrasting needs. Since the most commonly used window systems are based on a client/server paradigm, the first problem is the so called priority inversion [19]. Mainstream window systems (e.g., X11) execute graphical primitives in an order that is irrespective of the real-time priority of the tasks formulating the request. Thereby, real-time tasks can incur a blocking time from lower priority tasks, for which it is difficult or impossible to find an upper bound. Blocking times can be reduced by using appropriate scheduling

IEEE computer society

mechanisms but they are lower-bounded by the length non-interruptible graphical primitives. Another non-trivial problem is that, in order to provide real-time guarantees to the clients, we have to take into account that these requests have to be scheduled in the slots of CPU execution time allocated to the server. Finally, real-time policies are commonly regarded to reduce the system throughput and this is hardly acceptable when a window system is used to manage heterogeneous applications, which only in part have timing constraints.

A first remarkable attempt to cope with these challenging issues is called Artifact [18], and it was developed in the early nineties as part of the real-time Mach project. Even if the authors use a single thread to manage all real-time request (potentially introducing priority inversion), they alleviate the problem by restricting to a small set the graphical primitives usable by real-time tasks. In this way the priority inversion is limited, but the authors themselves concede that a more satisfactory solution can only be found using different approaches (e.g., a multithreaded approach).The authors also consider the problem of coordinated scheduling of server and client by creating *scheduling models* for client and server upon a connection request, subject to a global admission test. In a more recent proposal called DOPE [9] (based on the DROPS kernel),the authors use a time-triggered thread to refresh the widgets belonging to real-time applications. An interesting idea is the introduction of resource managers that map basic resources to higher level ones, e.g., DOPE maps CPU cycles and main memory to refresh bandwidth. A different viewpoint is taken in EWS, a window system built in the context of the EROS kernel [20]. In this case the authors make a strong point that a WS with fast graphical primitives may provide good real-time performance without the need for an adequate scheduling support. In our evaluation, this statement is arguably true only when the system is not heavily loaded by a *large number* of client requests.

The proposals reported so far have a very important commonality: they are based on research systems, built from scratch to the purpose of displaying issues of interest, or to show the effectiveness of a specific solution. Research on CPU scheduling followed a similar path in the past, and the development of specialised real-time kernels, aimed at demonstrating some novel technique or scheduling algorithm, has been popular for some years producing solutions such as Nemesis [16], Real-time Mach [24], Hartik [5], YARTOS [10]. However, the absence of a strong connection with main-stream technologies such as Windows or Linux has severely hindered maintenance and porting across different hardware platform, reducing the impact of these proposals and ultimately causing their obsolescence. Therefore, there has been a paradigm shift toward architectural constructions tacked on general purpose kernels (typically

Linux). As a result, the real-time performance of such general purpose operating systems as the Linux kernel has improved impressively [14], to a point where today Linux is considered as a viable solution for both real-time research and industrial embedded software.

In this spirit, we believe that research on RTWSs should also move from writing research systems from scratch to modifying commonly used solutions; for this reason, we propose to introduce limited changes to the X11 server to make it suitable to real-time applications. The first contribution of this paper is a very thorough analysis of the X server to clearly identify the scenarios in which it fails to provide an adequate support to real-time applications. To this end, we have constructed a tracing tool that allows us to expose the timing behaviour of the X-server in different workload conditions. The conclusion of the first part of this work is that the effects of the scheduling policy are indeed quite heavy on real-time performance, and their importance outweighs the problems generated by the length of the non-interruptible primitives (which is being reduced by the new generation of accelerated graphical cards). This experimental observation is in perfect accordance with a similar conjecture formulated in [6]. In this case, case the authors propose a modification of the X11 server to support fixed priority scheduling. In our evaluation, this solution is not sufficient to lend robustness to the temporal behaviour of such soft real-time applications as continuous media, for which, as noted above, temporal protection plays a prominent role. Therefore, we have built a solution based on a particular flavour of the resource reservation algorithms, the constant bandwidth server (CBS).

The rest of the paper is organised as follow: Section 2 precisely identifies the problem and provides an analysis of its causes based on our tracing tool; Section 3 describes our solutions and Section 4 presents the obtained results. The conclusion and the future possibilities are explained in Section 5.

2 The Problem

The window system traditionally used in Unix-like systems is based on a client-server paradigm, where an *X server* acts as a manager for the video and for the input devices (keyboard and mouse). The X server forwards input events to some *client applications* and executes the *requests* received from the clients drawing on the screen.

Unfortunately, the X server is not aware of real-time requirements of time-sensitive clients, its only design goal being to maximise the global throughput. Indeed, it is possible to observe that the refresh rate of each window decreases with the number of active windows. This is hardly acceptable for real-time applications, for which a fixed refresh rate (independent of the workload of the server) is required. For

instance, in a media player, the users perceives this effect as a slowdown of the movie whenever some clients require a heavy operation.

In terms of the real-time scheduling theory, this problem can be classified as a **priority inversion**. The video card is accessed by a resource manager (the X server) that serves the different requests in an order that is not dictated by their real-time priorities. The duration of this priority inversion can be very long (potentially unbounded). The practical consequence is that even if the CPU is not overloaded and all the applications are properly scheduled by the CPU scheduler, time-sensitive applications can still be unable to perform their graphical operations with the correct timing. For example, a media player is not able to play a movie at the correct frame rate even if it is scheduled with the highest priority on the CPU (or if the CPU load - as measured by some utilities like top - is low).

To expose the problem and study it, we used an X testing application, called ocbench [23]. Ocbench is a very simple application which periodically updates a spinning wheel on the screen, using a very small amount of CPU time (so, it generates a large number of X requests that must be served in a timely fashion, but it creates a very low CPU workload). The application simply consists of an infinite loop that sleeps for a fixed amount of time (10ms in our tests) and then refreshes the image on the screen, and it lends itself to an immediate performance assessment. Indeed, when ocbench undergoes the interference of other X applications the user immediately notices a slowdown of the spinning wheel motion. In order to have a quantitative evaluation of this effect, we instrumented the code, by inserting a call to gettimeofday() right before the execution of the redraw operation. So, the progress of the measured times can be used as an indicator of the performance of the X server. The ideal evolution of this quantity is given by the $k(a + T)$, where k is the activation number, $T = 10ms$ is the periodicity of the requests and $a \approx 2ms$ is the duration of the redraw operation. To introduce interference on the execution of ocbench, we used the x11perf application (a standard test for the X server performance) that creates a large volume of X requests.

Table 1 shows the average and the standard deviation of the differences between two consecutive time readings of the instrumented ocbench. In particular, the first row refers to an execution of ocbench on a lightly loaded X server, whereas in the second one we considered the disturbing effects of an instance of x11perf -getimagexy100 executing in background,.

As shown in the table, when the X server is not loaded the average distance between the start of two consecutive redraw operations is about $12ms$, and the variance is about $0.182ms$ (in close accordance with the ideal behaviour). However, when the X server is heavily loaded
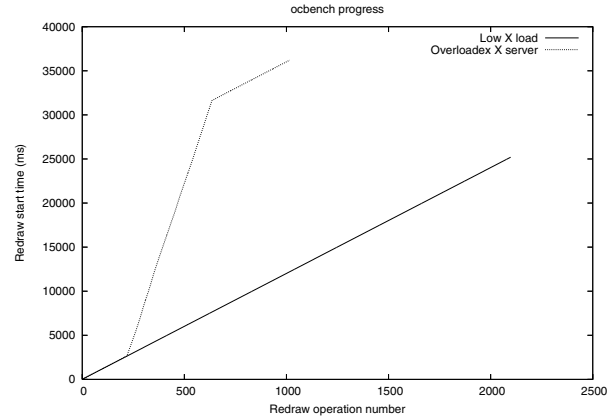
Figure 1. Evolution of the ocbench periods, with lightly loaded or overloaded X server.
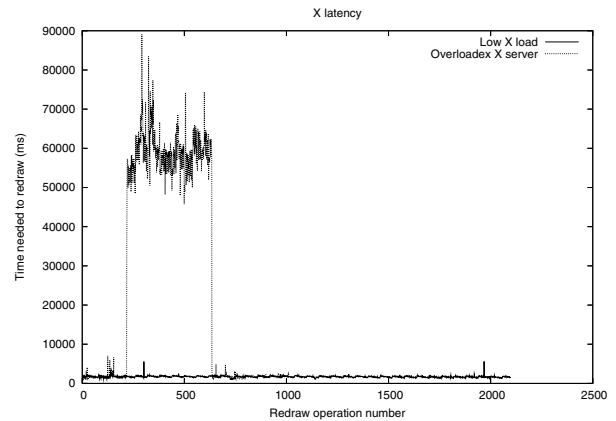
Figure 2. Evolution of the time needed by X to serve a request (X latency), with lightly loaded or overloaded X server.

by the x11perf application, the average distance increases and becomes much less stable (the variance increases to $28ms$). This behaviour is well depicted in Figure 1, which plots the evolution of the time readings as a function of the activation of ocbench. In case of a low workload the progress is exactly linear. In case of heavy workload, we have a piecewise linear plot. In particular, in correspondence with the 200th activation of ocbench, the slope becomes steeper (meaning a slower progress) due to the activation of x11perf. When x1perf is terminated (650th activation of ocbench) the slope becomes again equal to $12ms$.

To cast some light on this big variation in ocbench speed noticed in the second case, we measured the time needed by X to serve the redraw requests, inclusive of the

Workload level on the server	average ocbench period	standard deviation
low	12005 μs	182 μs
heavy	35606 μs	28475 μs

Table 1. ocbench periods with lightly loaded or overloaded X server.

execution time of the operation and of the scheduling delays. The result is plotted in Figure 2, in which the time required to complete the requests scales by orders of magnitude wen `x11perf` is activated.

2.1 Problem Analysis

As a first step to solve the priority inversion problems, we performed an in-depth analysis on the way the X server schedules the requests of its clients. To do this, we have written a patch of the X server that introduces a tracing mechanism to record the arrival of requests from the clients, their executions, and the times when each request is terminated. In particular, the following events are traced (each event is described by an *event type*, the *event time*, and the *client id*):

- **begin:** indicates the starting time for a trace;

- **end:** end of the trace;

- **creation:** a new client connects to the server;

- **destruction:** a client terminates, or disconnects from the server;

- **activation:** a new request from a connected client arrives (and is inserted into the queue of ready clients);

- **deactivation:** a request has been served, and is terminated;

- **dispatch:** the server starts to serve a request from a client (selected by the server's scheduler).

This tracer can be used to identify the sequence of events generated by the X server when it serves a client: for example, Figure 3 shows the trace of an instance of `ocbench` served by a lightly loaded X server. From the figure it is easy to see that the program periodically generates a burst of requests (corresponding to a redraw operation), which are immediately served by the X server (in about $2.5ms$ / $3ms$). As a result, `ocbench` can execute at a constant rate, and all the timing constraints are clearly respected. Note that each burst starts about $10ms$ after the end of the previous one, as expected (`ocbench` sleeps $10ms$ between two redraw operations).

Figure 4, on the other hand, shows a trace of `ocbench` when it competes with `x11perf` (that creates a heavy workload on the X server as discussed above). Right after

being started, `x11perf` runs a *calibration* phase, which in this trace ends at time $260ms$. During this phase, `x11perf` does not significantly affect the time execution of `ocbench`, which runs smoothly as in Figure 3. Around time 260, the calibration phase terminates, and `x11perf` starts to overload the X server: as a result, `ocbench` requests are not served in time. This phenomenon is easily detectable on the trace dedicated to `ocbench`; the bursts are no longer executed with regularity and some of them are delayed by a long time.

This effect has a clear explanation: since the scheduling mechanism adopted by X11 is a variant of the classical round-robin used in traditional time sharing systems, an application able to generated a large number of requests like `x11perf` is able to engage the server for arbitrarily long times starving other graphical applications. As shown next, the scheduling algorithm proposed in this paper allows us to radically alleviate this problem.

2.2 Interactions with the CPU Scheduler

The test applications used in the previous examples do not consume CPU time other than the one required for submitting request to X, so their real-time performance is only affected by the X scheduler (and the CPU scheduler contained in the kernel does not really matter, because the CPU load is low).

In more realistic situations things will likely be more complex, and there will be stronger interactions between the CPU scheduler and X scheduler: a request from an X client will only be served when the CPU scheduler selects the X server for execution, *and* the X scheduler selects the client. This means that performing real-time guarantees for X clients could potentially be quite difficult.

However, we believe that the complex system composed by the CPU scheduler, the X scheduler and the X clients' requests can be modelled as a hierarchical system. As a consequence, if both the X scheduler and the CPU scheduler provide predictable performance their combined effect can be analysed by applying the hierarchical scheduling theory [17, 11, 8]. This approach permits to simplify the analysis of the system, by considering the two schedulers in isolation, and composing their real-time guarantees. Since CPU schedulers have been studied at long in the past, and various theoretical frameworks for hierarchically composing scheduling guarantee exist in literature, in this paper we only focus on the behaviour of the X scheduler in isolation.
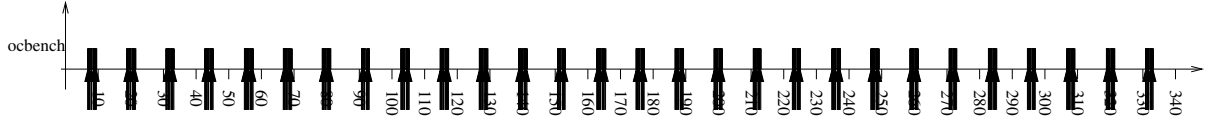
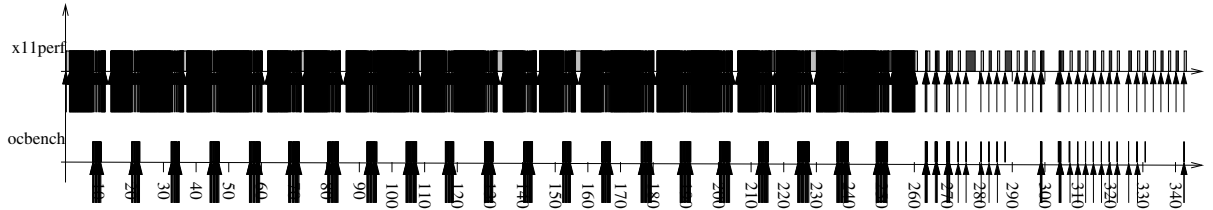**Figure 3. ocbench trace with non-loaded X server.**

Figure 4. ocbench trace with overloaded X server.

3 A Possible Solution

As shown in the previous section, the absence of an appropriate scheduling mechanism inhibits the use of X11 for real-time clients. On the other hand, fixed priority mechanisms based on classical real-time scheduling theory [12] like the one adopted in linux-SRT [6] have evident shortcomings when the task set encompasses both real-time and non real-time tasks. In particular, a design based on the worst case requirements of the tasks can be overly conservative. On the other hand, if we use a design based on the average resource requirements, a high priority real-time task consuming more than expected can cause deadline misses in other (unrelated) real-time tasks [4]. This problem is well known in the context of CPU scheduling, and a number of different solutions has been proposed. In particular, we based our work on a scheduling algorithm, called the Constant Bandwidth Server (CBS) that implements the *Resource Reservations* paradigm. In the development of this scheduler, we considered the X server as an *Open System* [7]. An Open System is a system in which applications dynamically enter and exit the system in an unpredictable (or hard to predict) way. Tasks (X clients, in this case) can dynamically be activated at any time and are characterised by variable execution times, so the scheduler cannot make any restrictive assumption on the characteristics of the task set.

Resource Reservations [13] have emerged as an effective technique to support time-sensitive applications on general purpose operating systems (GPOS). This technique provides support for time-sensitive applications by allowing the integration of classical real-time techniques, developed to meet timing constraints on real-time operating systems (RTOSs), with the general-purpose allocation strategies used on GPOSs. In particular, CPU reservations have been traditionally implemented by using a dedicated aperiodic server (the Deferrable Server [22]) to serve each reserved task [13, 15]. Unfortunately, this implementation strategy generates some scheduling anomalies when tasks block and unblock dynamically [22]. This particular problem has been solved by the CBS algorithm [1], which uses dynamic priorities to correctly cope with dynamic aperiodic arrivals. Therefore the CBS can provide a predictable QoS to both periodic and aperiodic real-time activities [2].

Since X requests are generally non periodic (for example, in Figure 3 we can see that requests arrive in bursts), the CBS appeared as a very natural choice for our reference implementation. However, the original version of the algorithm is fully preemptive. Thereby, it cannot be directly used in the X server (in which each request is substantially uninterruptible). For the sake of clarity, we will first briefly recall the original algorithm. Then we will review the most important features of the X server architecture (to discuss how our solution has been implemented). Finally, we will discuss the modification required to the CBS algorithm to adapt it to the X server.

3.1 The Constant Bandwidth Server

In the sequel an *X client* issuing a request to the server will be considered as a real-time task and referenced as τ_i; when receiving requests from multiple clients, the X server selects the request to be served according to some parameters and variables.

Each client τ_i is characterised by two parameters Q_i and T_i (we say that τ_i is associated to a reservation $RSV_i = (Q_i, T_i)$), meaning that it is *reserved* a time Q_i in a period T_i. According to the original (fully preemptive) CBS algorithm, when a request from client τ_i is served, the time needed by the X server for serving such request is accounted by decreasing a variable q_i called *budget*, and clients are scheduled based on their *scheduling deadlines* d_i^s (the client

with the earliest scheduling deadline is scheduled). When a request from client τ_i arrives to the X server at time t, we say that τ_i is activated. On the first activation of τ_i, q_i is initialised to Q_i and d_i is initialised to $t + T_i$, and when q_i arrives to 0 τ_i is said to be depleted. On depletion, two different behaviours are possible:

- the budget is immediately replenished to Q_i and the scheduling deadline is postponed to $d_i^s + T_i$ (so, the client remains schedulable). This is known as *soft reservation behaviour*

- the client is not schedulable until time d_i^s, when the budget will be replenished and the deadline will be postponed as above (so, the client cannot be scheduled until d_i^s). This is known as *hard reservation behaviour*

The CBS is guaranteed to execute respecting the real-time properties of the tasks inasmuch as the following condition is respected:

$$\sum_i \frac{Q_i}{T_i} \leq 1 \tag{1}$$

For a further discussion on the CBS algorithm and on its properties, the reader is referred to the original paper [1].

3.2 The X Server Architecture

The X server is a single thread application, in which a single flow of execution cyclically intercepts input events and receive clients' requests, selects the action to process, and processes it. While a multithreaded server (creating a thread per client) can easily delegate to the CPU scheduler (in the OS kernel) the selection of the client to be served, a single-threaded server like X must explicitly contain a scheduler for this purpose. On the hand, retaining the single threaded structure allows us to reduce the modifications required on the X server to implement the real-time scheduler and facilitates porting across the different versions of X.

The X server is logically structured in four layers - an OS dependent layer (OS), a device independent layer (DIX), a device dependent (DDX), and an Extension interface - and the scheduler is located in the DIX layer.

As said, the X server is implemented according to an event-based paradigm, with a main loop waiting for events, scheduling an event to be served, and serving it. There are three different kinds of events:

- connections from a new client;

- input events from the user (mouse click, ...);

- requests from an already connected client (clients' activations).

Events are served in a non-preemptable way, and the selection of the event to be served is performed by the scheduling function `SmartSchedule()`, which implements a variation of the round robin algorithm. So, when an event representing a request from client τ_i is selected (τ_i is scheduled), it executes until completion; moreover, to improve the throughput when τ_i scheduled it can execute a burst of requests (and not only one). In particular, a global variable called `isItTimeToYield` is used by the server to decide when new scheduler invocation are needed. In the standard implementation of X, `isItTimeToYield` is set when there are no more requests from a client (the client is deactivated), or when a maximum number of requests have been served.

3.3 Implementing the CBS on the X server

A modified (non fully-preemptable) version of the CBS scheduler has been implemented in the X server as an extension that can be enabled at compile time. When the feature is enabled, our implementation replaces the `SmartSchedule()` function with a CBS scheduler. First of all, we extended the information maintained for every client to store the CBS parameters and runtime variables:

- rt_period: the reservation period T_i;

- rt_deadline: the scheduling deadline d_i^s;

- rt_capacity: the current budget q_i;

- rt_maxcapacity: the maximum budget Q_i.

Then, we modified the X scheduler introducing a real-time queue (RTQ) ordered by scheduling deadlines, in which requests from clients associated to a CBS are stored. If there are requests in the RTQ, then the first one (i.e., the one having the earliest scheduling deadline) is selected, otherwise the original X scheduler is used.

Notice that since events are served in a non-preemptable way, the accounting can only be performed after serving an event, so q_i is updated when the request has been completely served. This means that after serving long requests q_i can become negative (if the maximum time C needed to serve a request was known in advance, we could consider q_i exhausted if $q_i < C$, but such assumption is not reasonable in an open system). If after performing the accounting $q_i \leq 0$, then

- $q_i = q_i + Q_i$, $d_i^s = d_i^s + T_i$, and the RTQ is reordered (soft reservation behaviour), or

- the client is removed from the RTQ, and will be re-inserted only at time d_i^s when budget and scheduling deadline will be updated as above (hard reservation behaviour)
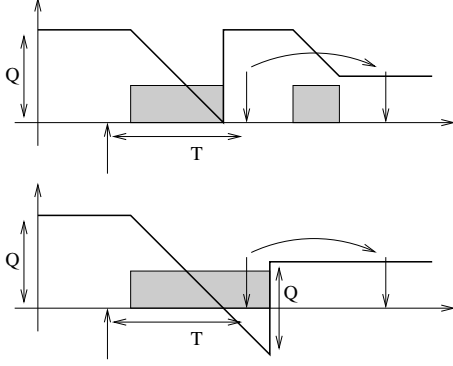
**Figure 5. Original and modified accounting and replenishment mechanisms.**

Figure 6. ocbench progress with lightly loaded or overloaded X server.

Note that since q_i can become negative, the replenishment is performed by setting $q_i = q_i + Q_i$, and not $q_i = Q_i$. Finally, when the scheduling deadline is postponed the RTQ must be reordered, and a different client is selected for service (note that this only happens after finishing to serve a request).

Figure 5 shows the effects of the modified accounting mechanism: the upper part depicts the behaviour of the original accounting and replenishment rules (for soft CBS) , while the the lower part shows the modified rules in action. When a new request arrives, it is assigned a deadline equal to the arrival time plus T; after some time the request is scheduled and its budget starts to decrease. In the original algorithm, when the budget arrives to 0 it is recharged to Q (remember that we are considering the soft incarnation of the CBS algorithm) and the deadline is postponed by T. Note that since the deadline is postponed, the client can be preempted (in the example, it is actually preempted, and is scheduled again only after some time). In the X implementation of the CBS, since the request is not preemptable it cannot be interrupted when the budget arrives to 0, and the request runs to completion. When it is finally served, the budget is negative, and is recharged by Q, postponing the deadline by T as above. Finally, note that the non-preemtpability of the requests can be modelled through the concept of *blocking time*: each request can cause a blocking time as long as the maximum time needed by X to serve it. So, if an upper bound B for the time needed by the X server to serve a request is known, it is possible produce an update of the admission test in Equation 1:

$$\forall i, \sum_{j=1}^{i} \frac{Q_j}{T_j} + \frac{B}{T_i} \le 1$$

(see [3] for more details).

Clients can manage their CBS parameters (Q_i, T_i) by using three new functions:
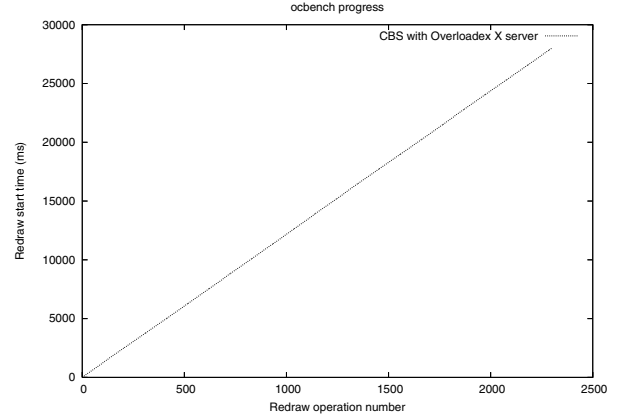
- `XRTInitialize()`: initialise the RT structure and check if this extension is installed;

- `RTSetProperty()`: transform the client in real-time (CBS) and set the reservation parameters

- `RTGetProperty()`: return some real-time information about the client

As a final remark, this scheduler could be plugged into the structure of X11 with a small effort. All these functions have been developed as X extensions strictly conforming to the guidelines provided by the Xorg foundation (that is, by properly extending the `xext` protocol and by implementing the functions in the Extension Layer) and without interfering with the normal functionalities of the window systems. Therefore, it is possible to rely on the hardware support for a plethora of graphic cards offered by the most commonly used X11 servers (Xfree86 and xorg) and to use the legacy applications without any porting effort (the API of the Window Systems has been totally unaffected).

4 Experimental Results

To test the effectiveness and the efficiency of the proposed solution, we performed an extensive set of experiments on a real implementation of our scheduler, considering different types of X clients (both real-time and not). All the experiments have been run on a standard PC based on an Intel core duo CPU at $1.66GHz$ equipped with an ATI Radeon X1300 graphic card and 1GB of RAM. The system is running Ubuntu Festy with a standard 2.6.20 Linux kernel, and the X server from current git.

4.1 Serving Time-Sensitive Applications

The objective of a first set of experiments was to verify that the CBS scheduler implemented in the X server properly addresses the problems exposed in Section 2. To this end, we considered the same situation depicted in Figure 1, in which a time-sensitive application (`ocbench`) is scheduled while an instance of `x11perf` overloads the X server and a measure of the system time is taken for each `ocbench` activation right before executing the redraw operations. Contrary to what we did before, the `ocbench` application is scheduled using a CBS server, both in the hard and in the soft version, with parameters $(3ms, 10ms)$. The result is reported in Figure 6 and the difference with the results displayed in Figure 1 is evident. In this case even after the activation of `x11perf`, which occurs after 200 `ocbench` cycles the time marks a perfectly proportionate progress in time. The speed of this progress is only affected by the choice of parameters for the CBS.

Since the amount of X time reserved to `ocbench` is enough for properly serving its requests, the hard and soft CBS schedulers provide the same performance, so the figure displays only one set of results. The correctness of the timing behaviour is confirmed by Figures 7 and 8, which represent the traces obtained when scheduling ocbench with a hard or soft CBS. As it is possible to see, reserving a correct bandwidth to the client allows the server to fulfill all requests with the proper timing.

4.2 Impact on Throughput

One of the problems encountered when applying real-time techniques in general-purpose systems is that they often have a bad impact on the throughput of non real-time applications. To show that the CBS implementation presented in this paper does not suffer from this problem, a second batch of experiments has been performed to gauge the effects of the CBS on the throughput of non time-sensitive applications. To this end, the throughput (number of operations per second) achieved by instance of `x11perf` served by a CBS has been measured (relying on the numbers reported by the `x11perf` program to compute the throughput).

First of all, `x11perf` has been scheduled through a soft CBS, obtaining an average throughput of 360 operations per second with a standard deviation of 1.73, while the throughput obtained using the standard X scheduler is 357 operations per second with standard deviation 2.28. We performed several experiments of this type obtaining consistent results: in all cases the CBS did not worsen the throughput (sometimes the results with the soft CBS were even better than the standard X11 scheduler [1]).

---

[1] This surprising result is probably due to the fact that scheduling
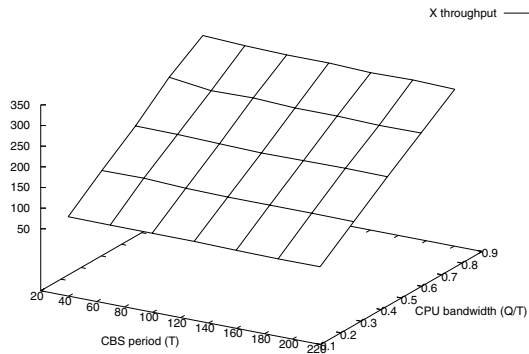
Figure 9. x11perf throughput when scheduled by a CBS with different parameters, in 3D.

Note that serving non time-sensitive applications with a CBS allows to have some degree of control on the applications' throughput. Hence, a third batch of experiments was designed to show the ability of the CBS to control the fraction of time devoted by the X server to serve requests from a specified client. In this case we used a hard CBS to schedule an instance of `x11perf` with different scheduling parameters. For each choice of parameters Q and T me measured the throughput achieved through 5 trials. The results (average and standard deviation) are shown in Table 2 and Figure 9. The achieved throughput increases proportionally to the bandwidth Q/T. It is worth noting that the performance obtained with the soft CBS was remarkably better (360 operations per second, as measured in the previous experiment). This is perfectly consistent with our expectations, since the hard CBS is used exactly to the purpose of allocating a hard bound to the time dedicated to the application (e.g., to avoid starvation of other applications running in background).

4.3 Scheduling a Media Player with the CBS

To show how the CBS can improve the performance of more complex time-sensitive applications[2], some of the previous experiments have been repeated using a media player. To this purpose, a media player based on FFMPEG and GTK/GDK has been ran together with an instance of `x11perf` (used to create a high load on the X server). The player has been configured for not skipping video frames,

---

[1] `x11perf` with a CBS gives him a higher priority over non real-time clients, such as the window manager

[2] remember that `ocbench` is a very simple application, designed to be only used as a benchmark

Figure 7. trace of an ocbench scheduled by a hard CBS, with overloaded X server.
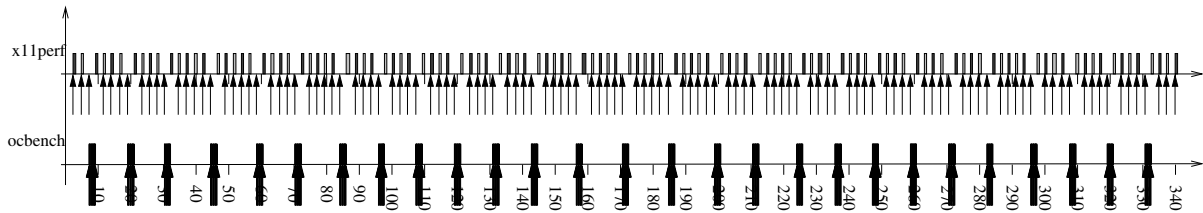

Figure 8. trace of an ocbench scheduled by a soft CBS, with overloaded X server.

and the Inter-Frame time (defined as the difference between the display times of two consecutive frames) has been used as a measure of the QoS perceived by the user.

Figure 10 shows the Inter-Frame times obtained when playing a video at 25 frames per second (fps) when using the standard X scheduler (this player will be referred as nrt player), and when serving the player with a properly dimensioned CBS (rt player). Until arond frame 110, the X server is not overloaded and all the frames from both the player instances are displayed in time (note that the Inter-Frame times are arond $40ms = 1/25$). Then, an instance of xllperf is started around frame 110, and overloads the X server causing a large increase in the Inter-Frame times experienced by the nrt player. The rt player instead, is not affected by the xllperf load and its Inter-Frame time remain stable all the time. During xllperf execution, the video frames from the nrt player are not displayed in time, and are queued by the X server; when xllperf stops, such frames are displayed at a high speed until the X queue is empty and the Inter-Frame times return to $40ms$.

5 Conclusions

In this paper, we analysed the performance of the X11 server when it is used to serve real-time requests, showing some anomalies caused by the priority inversions that stem from a substantially round-robin mechanism in managing a shared resource (the video card), which do not account for real-time priorities.

To solve this problem, we devised our own scheduling solution based on a CBS server. Advantages of our approach are: 1) the containment of priority inversion, 2) temporal isolation between the different applications, 3) easy
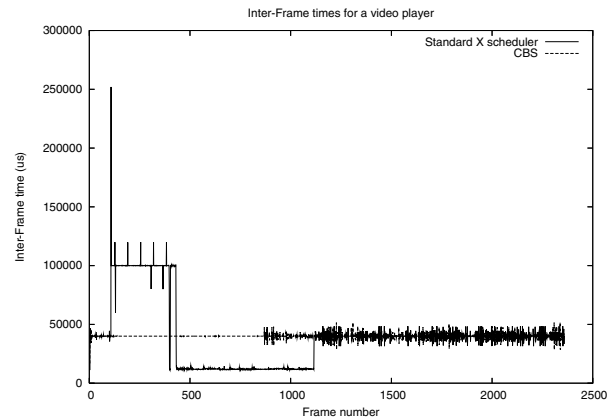

Figure 10. video player served by the standard X scheduler and by a CBS.

portability across different X versions, 4) negligible effects on the global throughput.

As a future work, we plan to study the interactions between the X scheduler and the CPU scheduler contained in the kernel, and to formally analyse the complex hierarchical system composed by the X server and its clients. As far as architectural aspects are concerned, we will test our solution on a wide class of applications to cover the whole gamut of graphical primitives and propose it as a full-fledged alternative to the standard scheduler.

	$Q = 1/6T$		$Q = 1/3T$		$Q = 1/2T$		$Q = 2/3T$		$Q = 5/6T$	
	Avg	StdDev	Avg	StdDev	Avg	StdDev	Avg	StdDev	Avg	StdDev
$T = 30$	67	1.14	132	3.64	193	2.40	264	2.3	318	2.48
$T = 60$	66.2	0.96	133	0.55	192	1.78	251	1.58	313	1.34
$T = 90$	66	0.55	128	1.51	189	2.04	253	1.30	310	1.48
$T = 120$	65.8	1.01	126	1.22	186	2.28	248	1.92	310	1.82
$T = 150$	63.7	1.12	126	1.34	187	2.28	248	2.49	306	2.30
$T = 180$	62.5	0.99	126	1.64	188	2.34	245	3.36	308	2.00
$T = 210$	62.8	1.04	125	1.08	187	2.30	246	2.60	305	4.1833

Table 2. x11perf throughput when scheduled by a CBS with different parameters.

References

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[2] L. Abeni and G. Buttazzo. QoS guarantees using probabilistic dealines. In *Proceedings of the IEEE Euromicro Conference on Real-Time Systems*, York, England, June 1999.

[3] T. P. Baker. Stack-based scheduling of real-time processes. *Real-Time Systems*, (3), 1991.

[4] G. Buttazzo, G. Lipari, L. Abeni, and M. Caccamo. *Soft Real-Time Systems: Predictability vs. Efficiency*. Springer Verlag, 2005.

[5] G. C. Buttazzo. Hartik: A real-time kernel for robotics applications. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1993.

[6] S. Childs and D. Ingram. The Linux-SRT integrated multimedia operating system: Bringing qos to the desktop. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, Taipei, Taiwan, 2001.

[7] Z. Deng and J. W. S. Liu. Scheduling real-time applications in open envirovment. In *Proceedings of the IEEE Real-Time Systems Symposium*, San Francisco, California, December 1997.

[8] X. A. Feng and A. K. Mok. A model of hierarchical real-time virtual resources. In *Proceedings of the IEEE Real-Time Systems Symposium*, page 26, Austin, Texas, 2002.

[9] N. Feske and H. Hartig. Dope - a window server for real-time and embedded systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 74–77, Cancun, Mexico, 2003.

[10] K. Jeffay, D. Stone, and D. Poirier. Yartos: kernel support for efficient, predictable real-time systems, 1991.

[11] G. Lipari and E. Bini. A methodology for designing hierarchical scheduling systems. *Journal of Embedded Computing*, 1(2):257–269, 2004.

[12] C. L. Liu and J. Layland. Scheduling alghorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1), 1973.

[13] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves for multimedia operating systems. Technical Report CMU-CS-93-157, Carnegie Mellon University, Pittsburg, May 1993.

[14] I. Molnar et al. Real-time linux wiki. http://rt.wiki.kernel.org.

[15] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa. Resource kernels: A resource-centric approach to real-time and multimedia systems. In *Proceedings of the SPIE/ACM Conference on Multimedia Computing and Networking*, January 1998.

[16] D. Reed and R. F. (eds.). Nemesis, the kernel – overview, May 1997.

[17] S. Saewong, R. R. Rajkumar, J. P. Lehoczky, and M. H. Klein. Analysis of hierarchical fixed-priority scheduling. In *Proceedings of the IEEE Euromicro Conference on Real-Time Systems*, Vienna, Austria, July 2002.

[18] J. E. Sasinowski and J. K. Strosnider. ARTIFACT: A platform for evaluating real-time window system designs. In *IEEE Real-Time Systems Symposium*, pages 342–352, 1995.

[19] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9), September 1990.

[20] J. Shapiro, J. Vanderburgh, E. Northup, and D. Chizmadia. Design of the eros trusted window system. In *Proceedings of the 13th USENIX Security Symposium*, 2004.

[21] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton. A proportional share resource allocation algorithm for real-time, time-shared systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, December 1996.

[22] J. K. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in hard-real-time environments. *IEEE Transactions on Computers*, 4(1), January 1995.

[23] W. Tarreau. The ocbench scheduler benchmark. http://linux.1wt.eu/sched.

[24] H. Tokuda, T. Nakajima, and P. Rao. Real-time mach: Toward a predictable real-time system. In *USENIX Mach Workshop*, pages 73–82, October 1990.