

Extending a Best-Effort Operating System to Provide QoS Processor Management

Hans Domjan and Thomas R. Gross

Departement Informatik
ETH Zürich, CH-8092 Zürich
hans.domjan@ethz.ch, thomas.gross@ethz.ch

Abstract. The benefits of QoS network features are easily lost when the end-nodes are managed by a conventional, best-effort operating system. Schedulers of such operating systems provide only rudimentary tools (like priority adjustment) for processor management. We present here a simple extension to a processor management system that allows an application to reserve a share of the processor for a specified interval. The system is targeted at applications with frequently changing resource demands or recurring, though non-periodic resource requests. An example of such an application is a network-aware image search and retrieval system, but other network-aware client-server applications also fall into the same category. The admission control component of the processor management system decides if a resource request can be satisfied. To limit the amount of time spent negotiating with the operating system, the application can present a ranked list of acceptable reservations. The admission controller then picks the best request that can still be satisfied (using the Simplex linear programming algorithm to find the best solution). If there are insufficient resources, the application must deal with the shortage. Any possible adaptation (if the accepted request was not the application's first choice) is left to the application. The processor management system has been implemented for NetBSD and been ported to Linux, and the paper includes an evaluation of its effectiveness. The overhead is low, and although reservations are not guaranteed, in practical settings the application almost always obtains the cycles requested.

1 Introduction

Many networks include either provisions or proposals to provide network services with defined QoS properties. Even if the QoS property cannot be guaranteed (in the sense that the network will ensure the properties even in the presence of catastrophic failures), nevertheless services with QoS properties aid tremendously in the construction of applications with defined timing behavior. E.g., a video conferencing system may use a bandwidth reservation to send voice and image data.

However, network QoS is just one aspect of providing true end-to-end QoS properties. The benefits of network QoS are easily lost if the end-node operating system does not cooperate. A conventional “best-effort” operating system (like many variants of Unix) provides only simple tools to assign appropriate processor cycles to applications with QoS network connections. Techniques such as boosting the priority or

over-provisioning of resources work if there is a single application, but cannot easily be extended to realistic scenarios [30]. Therefore, the operating system must allow an application to request some QoS. Such requests are usually for processor cycles, although other kinds of resources may be of importance as well.

This paper presents a simple extension to conventional best-effort operating systems which allows an application to request CPU resources for a time interval. Such requests are made at the time the demand is established, not in-advance [31]. This OS interface is geared towards applications that have recurring but non-periodic requests. An example of such an application is a client that presents the result of some query to the user. Queries are handled by a server and differ in the amount of data that have to be transferred to the client. If the client wants to present the result of the server-side query (e.g., a set of images or a video clip) within a fixed time (to offer predictable response time), then the amount of CPU resources needed by the server will be function of the volume and complexity of the data. Another type of applications that can benefit from QoS processor management are network-adaptive applications that are able to trade off one kind of resources (e.g., network bandwidth) for other resources (e.g., CPU cycles). E.g., network-aware adaptive applications can reduce their bandwidth requirements by transcoding (compressing) the data to be transmitted. However, such a transformation must be done within a specific time limit – the data must be transcoded when the communication subsystem is ready to transmit.

This paper is organized as follows: Section 2 presents the application model and the implications for a QoS CPU management system. Section 3 discusses the overall structure of the CPU management system. Section 4 contains the evaluation of this approach using three different real world applications. Related work is discussed in Section 5, and Section 6 contains the conclusions.

2 Application Model and Implications for CPU Management

2.1 Application Model

The proposed processor management system is targeted mainly at client-server type applications and provides short-notice, recurring and dynamic resource reservations at runtime. Applications *not* within this domain consist of multimedia applications (periodic resource requirements), real-time systems (static, hard resource requirements) and applications from the realm of video on demand or telecommunications, where resource reservation and allocation is typically made once at the beginning of a connection and remains in effect for the whole life time of the session.

In the model, the application must produce its result within a (user- or system-provided) time frame. The preparation of the result can be divided into one or several subtasks (which have their own time constraints inferred from the overall time frame), and there may be several algorithms (with differing resource requirements) at the application's disposal to carry out each subtask. Thus, reservation (and adaptation) decisions are not made only once at the beginning of the task, but can be recurring to take fluctuations of resource availability and demand into account.

If different algorithms impose different CPU requirements, then the application may choose the best algorithm that has a CPU requirement that can be satisfied by the proces-

processor management system. To cut down on the overhead of negotiating with the operating system, the application presents a ranked list of CPU requirements. This list contains the CPU demands for all possible algorithm options. The OS then decides which option is admissible, based on the option's resource requirements and overall system resource availability. The CPU requirements are expressed as a request for a number of cycles within a specific interval. The length of the interval is determined by an estimate of the corresponding subtask's length, and the interval may start either "now" or sometime in the future. The OS notifies the application which option can be admitted so that the application can take appropriate action. As long as the OS delivers the requested cycles in the interval, the application needs are satisfied. The OS thus is free to deliver all the requested resources at the beginning, evenly distributed, or at the end of the interval.

2.2 Implications for CPU Management

The precise resource requirements of the applications, as well as the number and timing constraints of their subtasks are unknown in advance; the subtask's resource reservations are recurring, though non-periodic in time. These details depend on many factors known only at run-time, like user input or the details of the available network QoS. Additionally, the application makes several reservation decisions while processing a task to take changes in resource availability into account, and may switch between best-effort mode for non time-critical administrative work and reserving mode for time-critical productive work. Furthermore the number of reserving applications competing for end system resources may vary, and even a single application may be multi-threaded. Finally, it is unrealistic to devote a whole host to support only one single application. The thread that produces a result should be given the CPU resources it needs while allowing other threads to proceed as far as possible. A CPU resource reservation in a best-effort OS is a good approach to address these characteristics because reservations provide a predictable QoS for applications that need it, yet accommodate conventional applications without modifications. Therefore a dynamic scheduler that accepts *reservation* requests at runtime, (re-)calculates the schedule on-the-fly and accommodates best-effort processes as well is a viable method for supporting this end-system QoS.

Reserving applications are driven by a model of their network and end system resource requirements, and estimates of future resource availability, both of which may not always be completely accurate. Thus, mismatches between predicted and actual resource consumption can be expected, and the processor management system should be able to handle such situations. *Over-reservation* is not a problem for a particular application, since it will receive the allocated resources anyway, but over-reservation by many applications degrades the overall end system throughput noticeably. *Under-reservation*, on the other hand, is more serious for an application and should be handled gracefully.

Since applications are allowed to use all end system operating system features, they may block on I/O or other events. The OS should be able to handle this case so that the reserved application that has blocked is later allowed to catch up the backlog without delaying other processes with reservations.

3 The R-Scheduler: A Pragmatic Approach to End System QoS

The abstraction provided by the R-Scheduler reflects the boundary conditions introduced in the previous section: To make a resource reservation, the application provides the OS with a vector of reservation requests. Each reservation request R_i consists of a pair (I, C_i) where I is an interval (defined by its *Start* and *End* time, where *Start* can be either “now” or in the future) and C indicates how many cycles this application wants to obtain in the specified interval. How and when the CPU is actually allocated within the interval I remains opaque to the application. The submitted reservation requests are sorted by the application in decreasing preference.

To determine which requests are feasible, the admission controller processes the vector by solving a system of linear equations that capture the resource requests for all time intervals, and then picks 0 or 1 of the individual reservation requests [10]. The application is informed about the admission controller’s choice and may then take an appropriate action, like executing the algorithm with a resource consumption that corresponds to the granted request. A process that has been granted a reservation request $R(I, C)$ is referred to as an R-process.

The resource demands C_i are often only estimates, and under-reservations may pose a problem to the application. To keep the R-Scheduler simple (and low-overhead), the R-Scheduler gives preference to R-processes with under-reservation over best-effort processes instead of a notification of the application and a possible re-negotiation of a reservation. On the other hand, in case of over-reservation, the R-process can yield no longer needed resources by means of a system call and make them available for new reservation requests. A detailed resource accounting scheme ensures that blocking R-processes are allowed to catch up their delay at the expense of best-effort processes, but not other R-processes.

The ability to obtain reservations is offered as an additional service to the user. Therefore all conventional, best-effort-type applications can be run unmodified on the system; only applications that want to take advantage of the reservations must be programmed accordingly. To ensure a minimal progress of best-effort applications in the system, only a pre-set fraction of the CPU is made available to R-Processes.

The R-Scheduling system has been implemented for the NetBSD [16] operating system and comprises of about 4200 lines of C code. In the actual kernel, only a few lines needed to be modified, mainly for the introduction of miscellaneous call-backs. In addition, the uneventful port of the R-Scheduler to Linux confirmed our assessment that adding such a scheduler to any reasonable operating system is straightforward.

4 Practical Experience

This section presents a comprehensive comparison of the QoS-enabled operating system (NetBSD) with the standard best-effort system. We report data for usage scenarios of three example applications, namely a resource-aware Internet server, a distributed image search and retrieval system, and an adaptive image decoder. All experiments were carried out on a 200MHz Intel Pentium Pro PC with 128MByte of RAM running NetBSD version 1.3 in an out-of-the-box configuration. We chose what is today

a low-end PC to demonstrate that it is possible also by modest platforms to provide application-beneficial QoS. In the experiments, at most 90% of the CPU are made available for R-processes, unless stated otherwise. The graphs show the mean values of five experiment repetitions, and error bars denote $\pm\sigma$.

Microbenchmarks have shown that the overall overhead of the admission controller and scheduler is modest even if the system is considerably loaded (below 0.7% with an average of four requests per second submitted and calculated).

4.1 Resource-Aware Internet Server

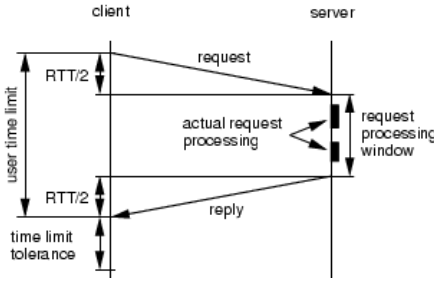
Description: The first example is a resource-aware Internet server. The clients submit requests to the server and expect a reply within a certain user time limit. Given that time limit, and the estimated RTT between server and client, the server has a request processing window at its disposal within which to generate the reply (Figure 1(a)). Assuming that the client prefers a prompt request rejection notification instead of a response later than the sum of the user time limit and a time limit tolerance, and that the server-side reply generation is a CPU-bound task, resource reservation in the server is a viable solution to achieve this desired quality of service. As an alternative to request rejection, the server may redirect the request to another server and thus use the resource reservation mechanism for predictable load balancing.

Evaluation: Upon arrival of a new request, the main server thread immediately forks off a new child process to handle that request. Under the best-effort OS, the child always tries to process the request. It either manages to finish before the end of the window (in this case a “success”-message is sent back to the client), or it exceeds the window, discards the result and sends back a “fail”-message. With the QoS-enabled OS, the child tries to allocate the CPU before processing; and in case of a rejection due to over-reservation it immediately sends back the “fail”-message to the client without any further attempt to process the request. Under both OSES, the main server thread can only be scheduled in best-effort mode due to the random and thus unpredictable request arrival.

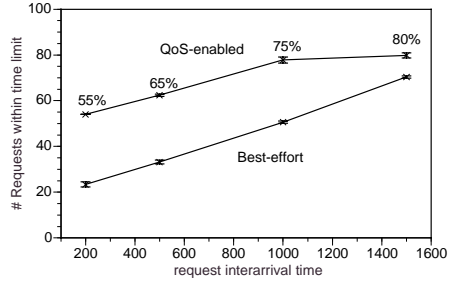
Both servers are subjected to four different request streams of 100 requests each (with exponentially distributed interarrival time with a mean of 200 ms, 500 ms, 1000 ms and 1500 ms respectively [1]; a user time limit of 5 sec [5], a time limit tolerance of 10%, a gamma distribution of RTTs according to [8, 20] and an exponentially distributed request processing time with a mean of 3000 ms (similar to [9]). The quality metric is the number of requests processed successfully within the time limit and tolerance.

Figure 1(b) shows the number of requests processed successfully for both OSES as a function of request interarrival time (for the QoS-enabled OS, the maximum percentage of the CPU allotted to reservations is given in the graph; see next paragraph for explanation). Performance increases with larger interarrival time because it leaves a larger overall timeframe for the server to process the identical requests. The QoS-enabled OS performs consistently better, but its advantage diminishes with less contention for resources (i.e., increasing interarrival time).

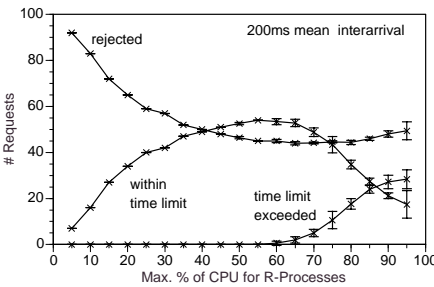
Figure 1(c) shows the request distribution as a function of the percentage of the CPU dedicated to R-processes, i.e., reservations, in the QoS-enabled OS. On one hand,



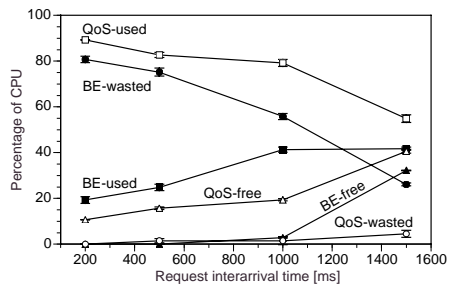
(a) Server Timeline.



(b) Server Performance Comparison.



(c) Performance under QoS-Enabled OS.



(d) Resource Usage Distribution.

Fig. 1. Resource-Aware Internet Server.

if only a small proportion of the overall resources (i.e., less than 20% of the CPU) is allocated to reservations, the QoS-enabled OS performs worse than the best-effort OS because the latter can use up to 100% of the CPU, whereas with such a configuration of the QoS-enabled OS a large percentage of the CPU must not be used by reservations, and thus too many requests are rejected. On the other hand, if too many resources (e.g., more than 70%) are allocated to reservations, the performance of the QoS-enabled OS starts to decrease and eventually becomes worse than that of the best-effort OS again. This behavior is due to the hybrid nature of the application where the main thread accepting connections is always running in best-effort mode, whereas the child thread actually processing the request is in reserved mode. Thus, if too many resources are allotted to (and used by) R-processes, the main thread has hardly any chance to run. The tasks of accepting the connection and forking off the child are delayed, and then there may not be enough time to provide the reply in time.

The percentage of resources that should be devoted to best-effort threads depends on the application scenario. These data, however, illustrate a subtle problem that may also be an issue for other processor management schemes that attempt to support network services with QoS properties. Administrative activities that establish the QoS properties

of a connection cannot, by definition, take advantage of any special treatment given to those threads that operate with QoS network connections.

Figure 1(d) shows the distribution of the CPU among “used” (processing of requests that eventually succeed), “wasted” (processing of requests that eventually fail) and “free” (leftover resources, e.g., for other reserving or best-effort applications) as a function of request interarrival time; the percentage is relative to the experiment duration (i.e., between arrival of the first and processing of the last request). We note that the QoS-enabled OS makes efficient use of the resources in the sense that it either allocates them for useful work, or leaves them unused; but does hardly waste them for requests that eventually fail.

Conclusions: We conclude from this experiment that the QoS-enabled OS provides a better service by up to a factor of 2.3 (in terms of successfully processed requests). However, this optimum is achieved only if neither too few nor too much resources are available for reservations. Furthermore, in this scenario where applications move back and forth between two service modes (best-effort vs. reserving), resources should be partitioned dynamically between the different classes depending on application behavior to achieve an optimal application performance. In our example, this optimum is achieved if the share for R-processes is as high as possible, but the fraction of requests exceeding their time limit is close to zero. This suggests an application feedback mechanism to the scheduling system where applications can indicate their optimal partitioning ratio. However, the generalization of this dynamic partitioning, especially if several applications with different usage scenarios are running on a single system, is subject of future research. Finally, the QoS-enabled OS allocates a high percentage of the resources to applications that need them.

4.2 Distributed Image Search and Retrieval System

Description: The second example is a distributed image search and retrieval system that attempts to adapt its behavior in response to changes in network resource availability [7, 29]. A client formulates a query for images, the system’s search engine identifies matching images, and the adaptive servers deliver the images in the best possible quality, considering network performance, system load, and a user-specified time limit.

The goal of the adaptation is to meet the user-specified limit on delivery time while maximizing the content quality of the images delivered. Content is correlated with size, so the system attempts to use its available bandwidth as well as possible. Therefore, while one thread transmits an object, concurrently a different thread prepares (transcodes) the next object(s) for transmission. To maximize the utilization of the available network bandwidth, the prepare thread should always have an object ready for transmission when the transmit thread can take another object. Therefore the application associates with each prepare step a deadline for completion that is derived from the model’s estimate of the duration of the concurrent transmission step.

This scheme of adaptation is not limited to this particular application but can be applied successfully to many network-aware applications with request-response communication. The core mechanisms have in fact been factored into a framework for network-aware applications [7].

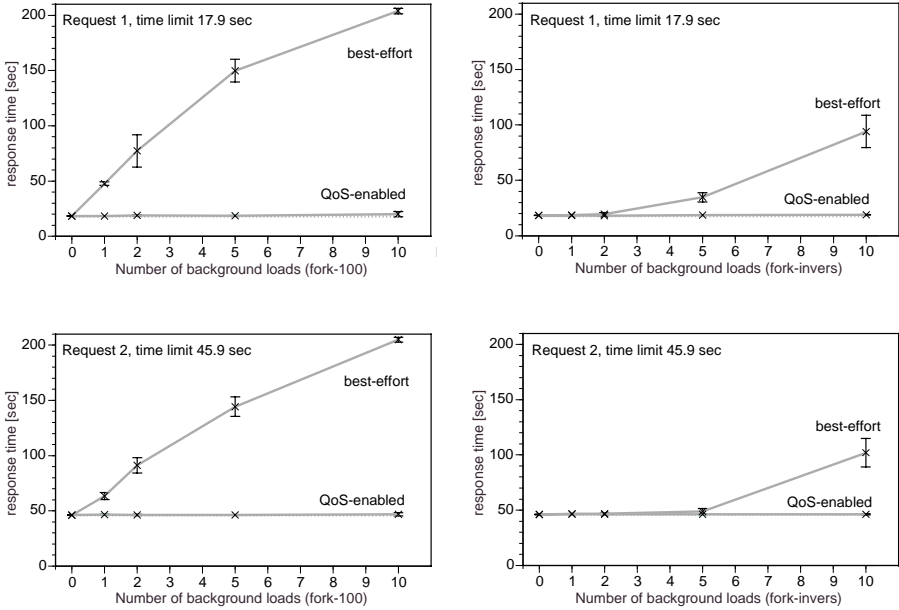


Fig. 2. Single Requests with Varying Background Loads.

Evaluation: For all experiments, the image server executes once under the best-effort OS, and once under the QoS-enabled OS. The same request for 25 images is used. Additionally, a varying number (0, 1, 2, 5, 10) of two different types of background loads is imposed: The aggressive load “fork-100” forks off a child every 100 ms and lets it run for 100 ms. The lighter load “fork-invers” forks off children with a more realistic lifetime according to [13]. The choice of forking background loads is justified since in a typical server scenario, new processes are created to handle client requests.

Experiment 1: One Server; Single Request. For this experiment, two different time limits are used: Request 1 specifies 17.90 sec (which yields overall image prepare costs corresponding to 26% average CPU usage), and Request 2 has a time limit of 45.9 sec (11% average CPU usage).¹

Figure 2 plots the response time of the image retrieval system, i.e., its ability to meet the user-specified time limit (dotted line), as a function of the number of actually imposed background loads. *Without* background load, there is no significant difference between the two OSes, because on one hand there are ample resources in the system, and on the other hand the QoS-enabled OS does not incur any noticeable system overhead. *With* background load, the situation is different. Under the best-effort OS, the server’s performance continually degrades as the number of background loads increases. The “fork-100” background load has a considerably more severe impact than “fork-invers”.

¹ The time limits are chosen so that for Request 1 a high image reduction ratio and for Request 2 a low reduction ratio is obtained. For more details, see [6].

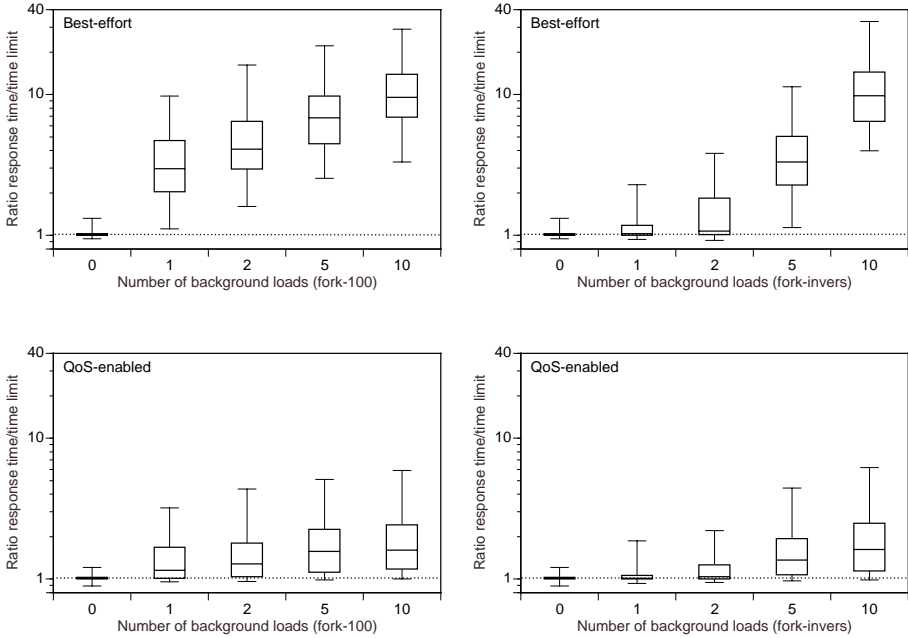


Fig. 3. Random Requests with Varying Background Loads; Distribution of Response Time.

Request 2 with the tighter time limit and thus higher CPU usage is more affected than Request 1. This is because the best-effort OS distributes the resources evenly among all competing applications. Under the QoS-enabled OS, however, the response time remains largely unaffected by both the kind and number of background loads, and additionally shows little variance. Thus the QoS-enabled OS is well able to provide a predictable service for reserving applications at the expense of best-effort applications.

An interesting observation is that using the given configuration, it would have taken 102 sec to transmit *all* the images *untransformed*, i.e., in their full size. The reason that it takes considerably more than 102 sec in the cases of the “fork-100” background load 5 and higher is twofold: On one hand, the simple cost model does not take into account that also the transmission of the images consumes little, albeit an under such background load not negligible amount of processing power. On the other hand, the simple CPU availability estimator fails in the case of this background load which consumes a disproportionate amount of CPU compared to pure compute-bound workloads. We conclude that under the QoS-enabled OS, the image server’s working range with the simple cost and load model can be extended to work also with aggressive background loads.

Experiment 2: Servers with Multiple Random Requests. In this experiment, the server is subjected to 50 randomly generated requests with an exponentially distributed interarrival time (mean 10 sec). The time limit is also exponentially distributed between

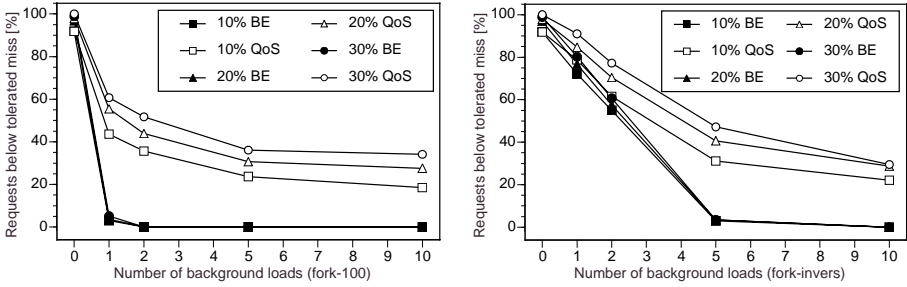


Fig. 4. Random Requests with Varying Background Loads; Percentage of Requests below Tolerated Time Limit Miss.

8.5 sec and 102 sec (mean 36.6 sec). For the given server configuration, these two values denote the two boundary cases where all images must compressed to minimal quality, or not reduced at all, respectively. This kind of requests produces an overall end system load with a dynamically varying number of concurrently running servers (from 0 to 6) and is designed to reflect a real server situation with bursts of requests.

Figure 3 shows the distribution of the aggregated ratio $2 \frac{\text{Achieved Response Time}}{\text{User Imposed Time Limit}}$ (with a log-scaled y-axis). The top, bottom and line through the middle of the box correspond to the 75th, 25th and 50th percentile, respectively. The whiskers on the bottom extend from the 10th and top 90th percentile. Without background load, the response time under the QoS-enabled OS is slightly better than for the best-effort OS because the former has, due to the reservations, a more precise knowledge about the application’s exact resource requirements and is thus better able to allocate resources at the right time to the appropriate application. As expected, performance degrades with increasing severity and number of the background loads. For the “fork-100” load, the QoS-enabled OS performs better by almost an order of magnitude.

In Figure 4 the “end-user” view is presented. This figure shows the percentage of requests that are handled on time, i.e., within the user-specified time limit plus a tolerances value of 10%, 20%, and 30%, respectively. Since the QoS-enabled OS does not provide hard guarantees, it may exceed a reservation’s time limit it by a few %; just reporting “success” or “failure” might present an inaccurate picture of the system’s capabilities. Note that the success to meet the time limit does not only depend on the end system OS but also on the application and its ability to adapt to changing network conditions.

Figure 4 shows that the best-effort OS is rarely able to allow the application to finish within the limit. With the “fork-100” background load, the failure is almost complete; for “fork-invers” there is a continuous decline as the load increases up to 5 background processes. A higher load implies that only a few percent of the requests succeed. With the QoS-enabled OS, however, between 43% and almost 18% (for a number of background loads between 1 and 10) of all requests do not exceed their time limit by more

² A value $X < 1$ thus means that the actual delivery was finished before the user-imposed time limit.

than 10%. If the user tolerates a time limit miss of 30%, between 61% and 35% of all requests are within this bound. For the “fork-invers” background load, the R-Scheduled server performs better than the best-effort scheduled one, although the difference is less pronounced than with “fork-100”.

Conclusions: We conclude from the experiments that the QoS-enabled OS is able to effectively shield applications from the detrimental effects of any kind of background load. Furthermore, the image server’s working range with the simple cost and load model can be extended to work with a larger variety of adversary loads. Finally, end user satisfaction (expressed in terms of time limit miss) is considerably better than with the best-effort OS despite the lack of hard guarantees.

4.3 Image Decoder

Description: The third example is an adaptive image decoder based on the Berkeley software MPEG-1 player [22]. An MPEG movie consists of three different frame types, namely self-contained I-frames, P-frames that depend on the previous and/or next I-frame(s), and B-frames that depend on the previous and/or next I- and P-frames. A simple way of adaptation to available end system resources is to decode the movie at one of three quality levels corresponding to the decoding of all frames (IPB), IP-frames, and I-frames only. The decoder determines the decoding resource requirements of a future movie sequence (whose length should be on the order of seconds) in the three different quality levels based on linear regression of the frame sizes [4] and submits this vector to the scheduler that decides, based on the available CPU resources, which level can be decoded. Subsequently, the frames are decoded into a buffer, from which they are displayed using any periodic scheduler with the specified movie frame rate.

Due to the large variability in decoding time characteristics as well as fluctuating end system resource availability, it makes little sense to reserve a certain bandwidth of CPU for the whole movie in advance (this might even be impossible in case of a live broadcast with undetermined end time), but resource reservations must continually be reconsidered.

Evaluation: The goal of this experiment is to show that the QoS-enabled OS provides effective support for dynamically adapting applications in a resource contention situation. The movie chosen for the experiments has a play time of 16.35 sec, and uses an average CPU bandwidth of 38% for decoding all frames (IPB), 17% for I and P frames, and 4% for the I frames alone. Under the QoS-enabled OS, the decoder allocates resources for an interval of 1 sec with the four levels IPB, IP, I and one I. Under the best-effort OS, the decoders run in different static adaptation configurations.

Table 1 shows the quality metrics “decoding time” and “percentage of decoded frames” for the different configurations. For two parallel decoders, both OSES perform equally well since there are enough resources to decode both movies. With increasing number of decoders, we note that to decode all frames, more time than the desired decoding time of 16.35 sec is needed. On the other hand, for a particular statically configured quality level, the desired decoding time is met, but at the cost of dropped frames.

Table 1. Image Decoder Results.

#Decoders	OS types (static configuration)	Overall CPU (%)	time (16.35 sec)	frames decoded (%)		
				I	P	B
2	Best-effort OS (IPB/IPB)	77	16.30	100	100	100
	QoS-enabled OS	75	16.09	100	100	100
3	Best-effort OS (IPB/IPB/IPB)	100	18.45	100	100	100
	QoS-enabled OS	95	16.19	99	98	76
	Best-effort OS (IPB/IPB/IP)	94	16.33	100	100	66
4	Best-effort OS (IPB/IPB/IPB/IPB)	100	24.37	100	100	100
	QoS-enabled OS	95	16.32	97	94	40
	Best-effort OS (IPB/IPB/IP/I)	98	16.75	100	75	50
	Best-effort OS (IPB/IP/IP/IP)	91	16.30	100	100	25
5	Best-effort OS (IPB/IPB/IPB/IPB/IPB)	100	30.66	100	100	100
	QoS-enabled OS	88	16.05	100	100	0
	Best-effort OS (IPB/IPB/I/I/I)	89	16.30	100	40	40
	Best-effort OS (IPB/IP/IP/IP/I)	95	16.44	100	80	20
	Best-effort OS (IP/IP/IP/IP/IP)	87	16.30	100	100	0
6	QoS-enabled OS	95	16.26	98	92	0
	Best-effort OS (IPB/IP/IP/IP/I/I)	95	16.50	100	67	17
	Best-effort OS (IP/IP/IP/IP/IP/I)	91	16.30	100	83	0
7	QoS-enabled OS	86	16.05	99	64	0
	Best-effort OS (IPB/IP/IP/IP/I/I/I)	100	16.62	100	57	14
	Best-effort OS (IPB/IP/IP/I/I/I/I)	90	16.34	100	43	14
	Best-effort OS (IP/IP/IP/IP/IP/I/I)	96	16.54	100	71	0

Conclusions: The main conclusions from this experiment are that the QoS-enabled OS supports applications in a graceful dynamic degradation of the quality if there is resource contention, without prior knowledge about end system utilization. Furthermore, the QoS-enabled OS permits applications to dynamically achieve a quality level comparable to static adaptation policies. The modifications necessary to turn the static image decoder into an adaptive one were modest, adding evidence that it is both feasible and worthwhile to change applications to use the features of the QoS-enabled OS.

4.4 Experiment Conclusions

The experiments in this section have shown that the QoS-enabled OS

- provides a superior service for applications (in terms of end user metrics) which is consistently visible in a number of usage scenarios with one or more QoS-aware applications, as well as a variation of background loads.
- allocates resources effectively to applications that need and can make use of them.
- extends the working range of adaptive applications having simple resource models.
- supports applications in dynamic adaptation in case of resource contention.
- can be integrated with minimal effort, and adds negligible runtime overhead to a best-effort OS.
- provides an easy-to-use interface for a variety of applications.

Furthermore, we have identified the issue of dynamic application-adaptive resource partitioning among best-effort and reserving resource usage classes as an issue of future research.

5 Related Work

CPU reservations are central to real-time systems that schedule a fixed set of periodic, independent, non-blocking tasks with known, constant execution times [15, 24, 25]. In contrast to real-time systems, the resource requirements of the target applications are dynamic, aperiodic and known in detail only shortly before the resources are needed, and not necessarily contiguous. Additionally, there are a dynamically varying number of best-effort processes and processes with reservations that have to be scheduled. Furthermore, the tasks may interact or block on I/O.

Schedulers for multimedia systems (like Processor Capacity Reserves [17, 18, 19], SMART [21], the Rialto Scheduler [14], ETI [3]) pose less stringent restrictions on the scheduled process set than real-time systems and can accommodate a dynamically changing number of processes with varying resource requirements, but they too are mainly targeted at supporting tasks with periodic resource demands. Furthermore, some schemes enforce adaptivity by dropping single periods — a behavior that is application-specific and does not fit the application model considered here.

Proportional-share schedulers like *lottery* and *stride scheduling* are resource allocation mechanisms providing efficient, responsive control over the relative execution rates of computations [12, 26, 27, 28]. In contrast to the proportional share model, the system presented here offers absolute, time-bounded resource reservations which can be contracted with the system in advance.

Conventional operating systems like UNIX [23] use dynamic priorities with decay-usage scheduling. This scheme gives high responsiveness for I/O-intensive applications and prefers them over long-running CPU-bound processes making it a good choice for interactive systems [2, 11, 16]. They do not, however, provide resource reservations or other means of predictable resource availability.

6 Conclusions

This paper presents the evaluation of a low-cost, pragmatic approach to processor management based on resource reservations. The R-Scheduler consist of an interface for applications to negotiate their future needs and a reservation-based scheduling scheme that prevents system overload. This R-Scheduler, which extends the notion of QoS from the network into the OS, co-exists with a best-effort scheduler and has been implemented with modest effort for NetBSD and ported to Linux.

Experiments with three different applications identify CPU scheduling as a key success factor for the effectiveness of a QoS-enabled OS. They show that such a QoS-enabled OS provides adequate support for resource-aware applications. Applications that attempt to limit their response time perform considerably better when using reservations than when execution is controlled by a traditional best-effort OS. Finally, the

effort required to make applications take advantage of the features of the QoS-enabled OS as well as to add those features to any reasonable operating system is modest.

As the importance of network-aware adaptive applications and differentiated services increases, operating systems are challenged to provide low-cost support for the CPU resource reservations. The scheduler presented here provides an approach that can be easily integrated into existing systems and can co-exist with current best-effort scheduling disciplines.

References

- [1] M. F. Arlitt and C. L. Williamson. Internet web servers: Workload characterization and performance implications. *IEEE/ACM Transactions on Networking*, 5(5):631–645, Oct. 1997.
- [2] M. J. Bach. *The Design of the UNIX Operating System*. Prentice Hall, 1986.
- [3] M. Baker-Harvey. ETI resource distributor: Guaranteed resource allocation and scheduling in multimedia systems. In *Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 131–144, Feb. 1999.
- [4] A. C. Bavier, A. B. Montz, and L. L. Peterson. Predicting MPEG execution times. In *Proceedings of the SIGMETRICS '98/PERFORMANCE '98 Joint International Conference on Measurement and modeling of Computers Systems, June 22-26, 1998; Madison WI*, pages 131–140, May 1998.
- [5] N. Bhatti, A. Bouch, and A. Kuchinsky. Integrating user-perceived quality into web server design. Technical Report HPL-2000-3, Internet Systems and Applications Laboratory, HP Laboratories Palo Alto, Jan. 2000.
- [6] J. Bolliger. *A framework for network-aware applications*. PhD thesis, ETH Zürich, Apr. 2000. No. 13636.
- [7] J. Bolliger and T. Gross. A framework-based approach to the development of network-aware applications. *IEEE Transactions on Software Engineering (Special Issue on Mobility and Network-Aware Computing)*, 24(5):376–390, May 1998.
- [8] R. L. Carter and M. E. Crovella. Dynamic server selection using bandwidth probing in wide-area networks. Technical Report BU-CS-96-007, Computer Science Department, Boston University; 111 Cummington St., Boston MA 02215, Mar. 1996.
- [9] J. Dilley. Web server workload characterization. Technical Report HPL-96-160, Hewlett-Packard Laboratories, Dec. 1996.
- [10] H. Domjan and T. R. Gross. Providing resource reservations for adaptive applications in a best-effort operating system. Technical report, ETH Zürich, Feb. 2001.
- [11] B. Goodheart and J. Cox. *The Magic Garden explained: the Internals of UNIX System V Release 4, an Open-Systems design*. Prentice Hall, 1993.
- [12] P. Goyal, X. Guo, and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 28-31 Oct. 1996; Seattle, WA, USA, pages 107–121, Oct. 1996.
- [13] M. Harchol-Balter and A. B. Downey. Exploiting process lifetime distributions for dynamic load balancing. In *Proceedings of the 1996 ACM SIGMETRICS Conference*, pages 13–23, 1996.
- [14] M. B. Jones, D. Roşu, and M.-C. Roşu. Cpu reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Proceedings of the Sixteenth ACM Symposium on Operating System Principles (SOSP-16)*, Saint-Malo, France, October 5–8, pages 198–211, Oct. 1997.

- [15] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association of Computing Machinery*, 20(1):46–61, Jan. 1973.
- [16] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. *The Design and Implementation of the 4.4BSD Operating System*. Addison-Wesley Publishing Company, 1996.
- [17] C. W. Mercer. *Operating System Resource Reservation for Real-Time and Multimedia Applications*. PhD thesis, School of Computer Science, Carnegie Mellon University Pittsburgh, PA 15213-3890, June 1997.
- [18] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: An abstraction for managing processor usage. In *Proceedings of the Fourth Workshop on Workstation Operating Systems*, Oct. 1993.
- [19] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 90–99, May 1994.
- [20] A. Mukherjee. On the dynamics and significance of low frequency components of internet load. *Internetworking: Practice and Experience*, 5(4):163–205, Dec. 1994.
- [21] J. Nieh and M. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proceedings of the 16th ACM Symposium on Operating System Principles, St. Malo, October 1997*, pages 184–197, Oct. 1997.
- [22] K. Patel, B. C. Smith, and L. A. Rowe. Performance of a software MPEG video decoder. In *Proceedings of the First International Conference on Multimedia; 1–6 August 1993; Anaheim, CA, USA*, pages 75–82. ACM, Aug. 1993.
- [23] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Communications of the ACM*, 17(7):365–375, July 1974.
- [24] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, Sept. 1990.
- [25] B. Sprunt, L. Sha, and J. Lehoczky. Aperiodic task scheduling for hard-real-time systems. *The Journal of Real-Time Systems*, 1(1):27–60, June 1989.
- [26] 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 17th IEEE Real-Time Systems Symposium*, pages 288–299, Dec. 1996.
- [27] C. A. Waldspurger and W. E. Weihl. Lottery scheduling: Flexible proportional-share resource management. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 1–11, 1994.
- [28] C. A. Waldspurger and W. E. Weihl. Stride scheduling: Deterministic proportional-share resource management. Technical Memorandum MIT/LCS/TM-528, MIT Laboratory for Computer Science, Cambridge, MA 02139, June 1995.
- [29] R. Weber, J. Bolliger, T. Gross, and H.-J. Schek. Architecture of a networked image search and retrieval system. In *Proceedings of the ACM Conference on Information and Knowledge Management (CIKM'99)*, pages 430–441, Nov. 1999.
- [30] L. C. Wolf. *Handbook of Multimedia Computing*, chapter Resource Management in Multimedia Systems, pages 891–912. CRC Press, Boca Raton, FL, USA, 1998.
- [31] L. C. Wolf, L. Delgrossi, R. Steinmetz, S. Schaller, and H. Wittig. Issues of reserving resources in advance. In *Proceedings of the Fifth International Workshop on Network and Operating System Support for Digital Audio and Video; April 19–21, 1995; Durham, New Hampshire, USA*, Apr. 1995.