

Managing Resource Reservations and Admission Control for Adaptive Applications

Hans Domjan and Thomas R. Gross
Departement Informatik, Laboratory for Software Technology
ETH Zürich, CH-8092 Zürich
Hans.Domjan@ethz.ch, Thomas.Gross@ethz.ch

Abstract

An important class of adaptive applications can trade off one kind of resources (e.g., network bandwidth) for requests of other resources (e.g., CPU cycles). They create new challenges for operating systems: their processor demands change rapidly based on external factors, and resource requests are recurring, though non-periodic. However, these applications share some of the characteristics of “soft real-time” tasks and are often resilient with regard to un- or under-availability of resources.

This paper presents a comprehensive approach to processor management for adaptive applications, the R-Scheduler. It co-exists with a best-effort scheduler and has been implemented for NetBSD and ported to Linux. The runtime costs of admission control and scheduling are modest (below 1%). For realistic usage scenarios, the R-Scheduler allows the application to meet its time limits, whereas the traditional (default) best-effort scheduling discipline fails to allocate the CPU resources effectively.

Keywords: *Processor Scheduling, Operating Systems, Resource Management, Adaptive Applications, Network-Aware Applications.*

1. Introduction

Adaptive applications have lately received considerable attention. Many applications that involve the Internet have to deal with the wide range of possible network performance: when connectivity is good, the application provides rapid responses, but congestion forces the user to wait. An adaptive application is able to tradeoff network resources (bandwidth) with local CPU cycles: If there is congestion, the application transcodes (compresses) the data that must be transmitted. A network-aware adaptive application is

thus able to provide predictable service (i.e., predictable response time) over a much larger range of network conditions, but such adaptivity may come at a cost in content quality.

Although the network conditions may drive in this scenario the adaptation decisions, the processor of the host system plays a crucial role, because adaptive applications trade off local computation (e.g., transcoding) for network resources (e.g., bandwidth). Therefore such applications create new challenges for schedulers in operating system: the CPU demands of these applications change rapidly (based on external factors, such as bandwidth availability) and the demands are non-periodic. On the other hand, the applications exhibit a high degree of resiliency, similar to “soft real-time” tasks. If for some reason the application is granted only a fraction of the desired CPU resources to perform a transcoding step, then such a “failure” is not catastrophic. The quality of the delivered content may be lowered, but most network-based applications are able to tolerate some degree of network bandwidth variation. Such tradeoff situations may also occur without a network: consider a system that renders scenes or replays a movie that is read from some storage device. If there is a shortage of CPU resources, adaptation through quality reduction allows the system to maintain a smooth flow of images.

Such adaptive application may also have some freedom in picking the order of processing objects, or they can choose different algorithms for a computation (e.g., transcoding or rendering). If there are options, the application presents the OS with a ranked list of CPU resource requests (corresponding to the costs of the operations) and allows the OS to satisfy *one* of these requests. The order of the requests in the list implicitly reflects the application’s preference. The ability of adaptive applications to decide at runtime, based on resource availability, which processing option to take next provides a degree of flexibility that is not found in other situations.

There exist several approaches to implement adaptivity; adaptivity may be integral to the application [25] or may be

⁰0190-3918/01 \$10.00 © 2001 IEEE

provided by a transcoding proxy [7]. This paper concentrates on the OS support that is needed for adaptive applications. The solution presented here consists of an admission controller, a scheduler, and a wide interface between application and admission controller. It has been implemented for NetBSD and ported to Linux.

2. Processor demands

We start with a brief summary of an exemplary network-aware adaptive application [25] to illustrate the OS mechanisms and concepts for supporting adaptive applications.

2.1. Application example

The example application is a distributed image search and retrieval system that attempts to adapt its behavior in response to changes in network resource availability [4, 25]. 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 core mechanism of the server is a software feedback loop that tries to bridge the gap between an estimated delivery time and the time left for the image transfer by making appropriate adaptation decisions (e.g., transcoding images, retrieving a different version). The server-initiated adaptation decisions are driven by a *model* of the expected delivery time [4], which includes estimates of future network bandwidth [5], the size of the transcoded images, and the CPU costs of transcoding steps (which depend on the size of the object, coding format, compression factor, and processor capabilities).

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. Transmission and preparation are controlled by a decision phase driven by the model mentioned above. 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 current transmission step.

This scheme of adaptation 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 [4].

2.2. Application and operating system interaction

In our model of adaptive applications, the application must produce its result (*delivery of all requested images*) within a (user-provided) time frame. Upon transmitting an image, the application determines the resource needs and availability for the transmission of the next image (*model evaluation*). If more data must be transmitted than there is bandwidth capacity, the application can use the CPU to transcode some objects. Thus, adaptation decisions are not only made once at the beginning of the task but are made repeatedly (*upon transmission of an image, or more frequently*) to take fluctuations of resource availability and demand into account.

Adaptation decisions are made at so-called *adaptation points (AP)*. A new AP is reached after the previous subtask completes. At every AP, the application must estimate the availability of network and end-system resources and determine its resource needs until the next AP (*the estimated completion of the next transmit phase*) and must then choose one of the (potentially multiple) options (*transcoding algorithms, choices of image to transcode*) to complete the next subtask.

When the application identifies options with different CPU requirements, then a "good" option is the one that can actually be realized given the availability of CPU resources. Therefore the CPU requirements of all possible options (*transcoding algorithms, choice of image to transcode*) are presented to the OS. The OS then decides which option is admissible, based on resource requirements and overall 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 next AP. The OS notifies the application of the option it has chosen, so that the application has the opportunity to take appropriate action. As long as the OS delivers the requested cycles in the interval, the application needs are satisfied.

2.3. Implications for CPU management

The precise resource requirements of adaptive applications, as well as the number and interarrival spacing of the APs are unknown in advance; APs are non-periodic in time. These details depend on many factors known only at runtime, like the transmission order for a sequence of objects, or the (fluctuating) bandwidth. Additionally, the application makes several adaptation decisions while processing a task to take changes in resource availability into account. Furthermore the number of adaptive 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 the prepare activity of

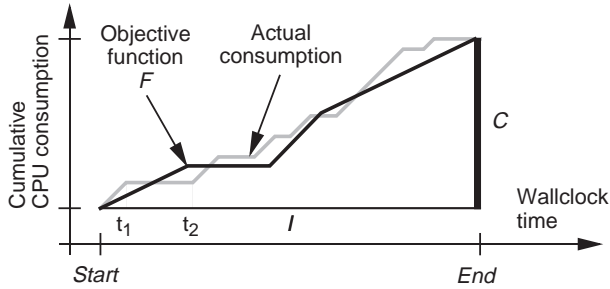


Figure 1. A reservation $R(I, C)$.

only a single server. The prepare thread should be given the CPU resources it needs while allowing other threads to proceed as far as possible. A CPU resource reservation is a good approach to address the characteristics of the prepare step because reservations prevent overbooking. Since several parameters (resources required, future bandwidth, time till next AP) are based on estimates, guarantees would not be appropriate. So a dynamic scheduler that accepts *reservation* requests at runtime and (re-)calculates the schedule on-the-fly is a reasonable compromise between the conflicting demands.

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

Last but not least, both best-effort processes as well as adaptive applications with reservations should be supported within one single system.

3. A comprehensive solution

The abstraction provided by our processor management system reflects the boundary conditions introduced in the previous section: At an AP, the application provides a vector of reservation requests. Each reservation request R consists of a pair (I_i, C_i) where I is an interval and C indicates how many cycles this application wants to obtain in the specified interval. The interval is defined by its *Start* and *End* time, expressed in wallclock time, and C is expressed in μsec , with $C < \alpha(\text{End} - \text{Start})$, as shown in Figure 1. α denotes the fraction of the overall CPU dedicated to processes with reservations. The submitted reservation requests are sorted by the application in decreasing preference.

This request vector is processed by the admission controller (Section 4.2), which picks 0 or 1 of the individual reservation requests. The application is informed about the admission controller's choice and may then take appropriate action (like executing the transcoding algorithm with a resource consumption that corresponds to the granted re-

quest). A process that has been granted a reservation request $R_k(I_k, C_k)$ is referred to during the interval I_k as a process with a valid reservation R , or as an R-process for short.

The semantics of a granted reservation $R(I, C)$ are as follows: Between *Start* and *End*, at least $C \mu\text{sec}$ of CPU time is provided to the holder of the reservation. How and when the CPU is actually allocated within the interval I remains opaque to the application. The scheduler is thus free to, e.g., allocate all C cycles at the very beginning, or at the very end of the interval, or to distribute the allotment over the whole interval. The resource delivery process can thus be modeled by using a monotonic objective function for cumulative resource consumption, as shown in Figure 1. To allow for this flexibility, R-processes must be always runnable during the complete interval for which they obtained a reservation. Section 4.1 presents a solution to the problems caused by processes that voluntarily sleep or block.

Each R-process can hold one reservation. Additionally, a $1 \rightarrow N$ relationship between a single (granted) reservation and multiple R-processes is allowed, e.g., for piped processes that jointly execute a task.

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), we do not provide for a notification of the application and a possible re-negotiation of a reservation. Instead, the R-Scheduler gives preference to R-processes with under-reservation over best-effort processes. On the other hand, in case of over-reservation, the R-process can yield no longer needed resources through a system call and make them available for new reservation requests.

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; only applications that want to take advantage of the reservations must be programmed accordingly.

4. R-Scheduler design and implementation

4.1. R-Scheduler

The R-Scheduler uses an *objective function for cumulative resource consumption* in conjunction with a *reactive scheduling discipline*. Figure 1 depicts an objective function (black line) and the actual resource consumption (gray line). We use compositions of linear functions to simplify both the R-Scheduler and the admission controller that calculates the objective functions. All F must fulfill the equation $F(\text{End}) - F(\text{Start}) = C$, and the gradients of each part of F must be $\leq \alpha$. No additional restrictions are placed on F ; F can be composed out of an arbitrary number of parts.

The *cumulative resource consumption* of an R-process is expected not to fall below its objective function F but

can precede F if there are ample resources available, like between $Start$ and t_1 in Figure 1. The *reactive scheduling discipline* takes action if the cumulative resource consumption of one of the R-processes has fallen below its F . It then arranges those R-processes to be scheduled immediately, as happens at time t_2 in Figure 1. Otherwise, the scheduling of all processes is left to the built-in best-effort scheduler.

If an R-process RP sleeps or blocks and becomes runnable thereafter, it has the largest backlog relative to its objective function F when compared to other R-processes (which were able to run meanwhile). Therefore the R-Scheduler would keep selecting RP , but this decision would prevent those other R-processes from running, thus causing them to miss their reservations. To avoid this problem, a detailed resource utilization measurement scheme is used that accounts separately for actual CPU usage (*OnCPUTime*), block time (*BlockTime*, involuntary preemption or being blocked on I/O) and sleep time (*SleepTime*, voluntary CPU yield). This scheme ensures on one hand that an R-process that voluntarily yields its resources has no right to reclaim them later at the cost of other R-processes. On the other hand, if an R-process has been involuntarily preempted from running, it is given the chance to catch up its backlog later if possible at the expense of best-effort applications, but *not* other R-processes.

The R-Scheduler is invoked periodically¹ and calculates for every R-process RP_i the value $\Delta_i = F_i - (OnCPUTime_i + SleepTime_i) - BlockTime_i$. A large Δ_i thus corresponds to a large backlog of RP_i relative to its objective function F_i ; a $\Delta_i < 0$ means that RP_i has a workahead relative to its F_i . Note that we account for the sleep time as if the process was running during that time.

To select the next eligible R-process, the list of all processes is traversed in decreasing order of Δ_i until either a runnable one is found or $\Delta_i < 0$. If, during the traversal, a blocked process is found, its *BlockTime* is incremented by the R-Scheduler period. If a runnable R-process is found, it is made eligible for immediate execution. It is then executing until either it blocks or is preempted, or until the next invocation of the R-Scheduler decides to schedule another R-process.

If, during the first traversal of the list of all R-processes, no eligible R-process is found, then for every R-process RP_i its Δ_i is recalculated as $\Delta_i = F_i - (OnCPUTime_i + SleepTime_i)$, and the above outlined search step is repeated with the new Δ_i to find R-processes that want to catch up a delay that stems from an involuntary preemption. If as a consequence an R-process obtains the CPU, its *BlockTime* is decremented by its actual CPU usage.

If there are still no R-processes found, then the schedul-

¹In the current implementation every 50 ms for a good trade off between minimizing scheduling overhead and maximizing reservation accuracy.

ing of *all* processes, i.e., both R-processes as well as best-effort processes, is left to the built-in best-effort operating system scheduler.

This accounting scheme ignores the processing costs for interrupts. Other researchers have already proposed solutions to this problem [2], which is not severe for our application domain (less than 2% overhead).

The objective functions bear some resemblance to value functions in real-time systems as introduced by Locke [13]. However, the objective functions used in the R-Scheduler express the cumulative resource consumption a process is to follow, whereas Locke's value functions describe the contribution of completing a job by its deadline to the overall system value as a function of elapsed job execution time.

4.2. Admission control and objective functions

The admission controller determines whether a new request for a reservation can be satisfied along the already granted ones, and it calculates and optimizes the objective functions of both the new and the previously granted reservations.

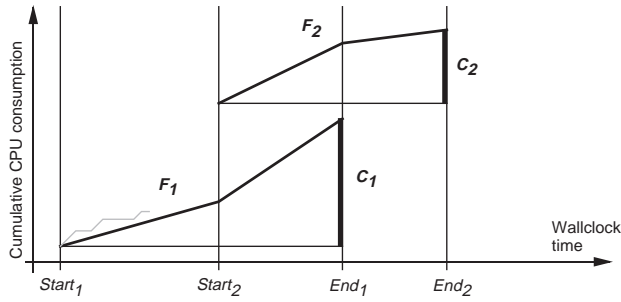
Its operation is best described using an example. Figure 2(a) shows the granted reservations $R_1(I_1 = [Start_1, End_1], C_1)$ and $R_2(I_2 = [Start_2, End_2], C_2)$ of two R-processes RP_1 and RP_2 , together with their objective functions F_1 and F_2 . RP_1 has already started its work, and its actual resource consumption—as denoted by the grey line—is ahead of the objective function F_1 .

At wallclock time “now”, a new request $R_3(I_3 = [Start_3, End_3], C_3)$ arrives, and the admission controller must decide whether this new request is satisfiable along the already granted ones R_1 and R_2 (Figure 2(b)). The union of all intervals $I = \bigcup I_i$ is divided into *segments* along the $Start_i$ and End_i times of *all* reservations, i.e. the granted ones and the new request under consideration, as shown by the vertical lines. For every reservation R_i , its objective function F_i is composed of linear functions with individual per-segment gradients. For each reservation $R_i(I_i, C_i)$, there are thus segments $S_i = \langle s_{i_1}, s_{i_2}, \dots, s_{i_{j(i)}} \rangle$. If $T_s(s_{i_k})$ denotes the start time of segment s_{i_k} and $T_e(s_{i_k})$ its end time, there is the natural constraint that $\forall i : T_e(s_{i_k}) = T_s(s_{i_{k+1}})$, $1 \leq k < j(i)$ and $T_e(s_{i_{j(i)}}) = End_i$. If $Start_i < now$ (as is the case for RP_1 in the example), we have $T_s(s_{i_1}) = now$, and $T_s(s_{i_1}) = Start_i$ otherwise. Thus only segments “in the future” are considered when recalculating the objective functions.

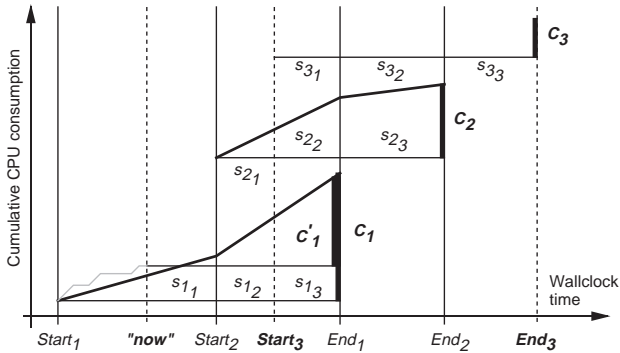
When processing a new request, the admission controller tries to recalculate all gradients under the following boundary conditions:

For all segments with the same start time, the sum of the gradients of the objective functions within that segment must be smaller than α :

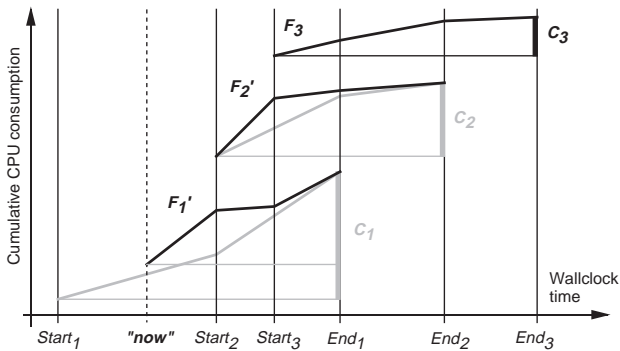
$$\forall k, l : \sum_{\forall i, j: T_s(s_{i_j}) = T_s(s_{k_l})} Gradient(s_{i_j}) \leq \alpha \quad (1)$$



(a) Two granted requests



(b) Newly arriving request R3



(c) New request R3 admitted, all objective functions recalculated

Figure 2. Admission control.

Equation 1 ensures that—at any time—the scheduling system does not contract out more CPU resources than are available.

For each reservation $R_i(I_i, C_i)$ this equation must hold:

$$\forall i: \sum_{k=1}^{j(i)} \text{Duration}(s_{ik}) \times \text{Gradient}(s_{ik}) = K_i \quad (2)$$

Equation 2 ensures that each R-process is granted its re-

source request by the end of its interval. For a reservation R_i that has not yet started its interval (i.e., $\text{Start}_i > \text{now}$), we let $K_i := C_i$ (Reservations R_2 and R_3 in Figure 2(b)). For a reservation R_i that has already started its interval (i.e., $\text{Start}_i < \text{now}$), we let $K_i := C'_i$, where C'_i is equal to the not-yet delivered part of C_i . This is the case for R_1 in the example, thus a potential workahead of R_i is accounted for when recalculating the objective functions.

The above boundary conditions, applied to all reservations (i.e., the granted ones as well as the new one under consideration), yield a set of in/equations. These in/equations may not have a solution, and if a solution exists, it may not be unique. The Simplex Linear Programming method [18] solves such a set under the additional condition of maximizing an arbitrary function. We maximize the overall utilization between “now” and the next (future) End (End_1 in Figure 2(b)) since a new request may arrive at that point from the R-process that held the just expired request, and we want to give the R-Scheduling system as much freedom as possible by “using up” whatever resources it can until then. Figure 2(c) shows the final situation with the recalculated objective functions of all RP_i after the R-request R_3 has been admitted.

Since by the API definition a request vector can be submitted, the above algorithm is carried out for each of the requests in the vector until a satisfiable reservation is found or all requests have been examined. Using linear programming in the admission controller of an operating system may raise concerns about response time, but as is shown in Section 5, today even low-end processors provide enough computing power to make this solution practical.

4.3. Admission policy

The admission controller decides solely on a “technical” basis whether to admit a new request or not. If several processes are competing for reservations, one of them may monopolize the processor and starve other processes.

To avoid this unwanted situation, an admission policy module is consulted before executing the actual admission controller. The policy module has knowledge of how many processes are competing for reservations (processes can register a (long) time interval during which they will request several (shorter) reservations), and possibly the request history of each process. Based on that information, it can remove those requests from the reservation vector that do not match the current admission policy.

4.4. Implementation details

The R-Scheduler and admission controller are implemented as a loadable kernel module for the NetBSD [14] operating system. A user-level library serves as the application

interface and performs the actual system calls into the R-Scheduler. Together with a standard textbook implementation of the Simplex Linear Programming package, the whole implementation comprises about 4'200 lines of C code (including whitespace and extensive comments).

The built-in scheduler is modified to accommodate two new priority levels above and in addition to the best-effort priorities, namely a “very high” and a “high” priority. Both levels are above the best-effort priorities and cannot be reached by those processes; additionally, priority aging for these two priority levels is disabled. The “very high” priority is used for R-processes selected by the R-Scheduler. The “high” priority is used for R-processes that have under-reserved resources and now are given preference over best-effort processes.

Whenever the built-in scheduler is called, it first checks whether the R-Scheduler must be called before doing its own work. If so, the R-Scheduler is called and, if it selects an R-process *RP* to run, adjusts *RP*'s priority to the “very high” level before allowing the built-in scheduler to continue. This step ensures that the built-in scheduler will select *RP*. Deselecting an R-process works by re-setting its priority to a value in the range used for best-effort processes.

In the actual kernel, only a few lines were modified, mainly for the introduction of a call-back that invokes the R-Scheduler from the built-in scheduler. Other modifications consist of disabling the process aging for R-processes and of adding up-calls to the system calls `exit` and `fork`.

5. Evaluation

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 the R-Scheduler works also on modest platforms. For all experiments, at most 90% of the CPU are made available for R-processes ($\alpha = 0.9$).

5.1. Synthetic microbenchmarks

Two experiments with different workloads are run: The first experiment simulates a lightly loaded system with few requests for reservations; the second experiment simulates a highly loaded system with many reservation requests.

The first workload uses random requests; each request vector consists of five requests. The length of the request intervals *I* is uniformly distributed between 0.5 and 20sec.

The second workload (infinitely) replays a trace captured from the image server of the adaptive application described in Section 2.1. A run consists of a total of 25 reservation requests. The first request submits a vector of 25 options, the second one a vector of 24 reservation requests, and so

Workload	Random requests		Image server trace	
	μ	σ	μ	σ
Experiment Duration	3687sec	2.22	3608sec	0.49
Requests submitted	1554	72	16754	88
Requests granted	1452	63	15858	316
Over/under-reservation ($X = \text{delivered} - \text{requested}$)				
$X < -50ms$	0.00%	0.00	0.04%	0.01
$-50ms \leq X < 0$	8.10%	0.92	8.81%	1.94
$0 \leq X < 50ms$	91.89%	0.92	91.14%	1.93
$50ms < X$	0.00%	0.00	0.00%	0.00
Relative overhead:				
– Admission controller	0.102%	0.014	0.574%	0.019
– R-Scheduler	0.023%	0.001	0.026%	0.001
– Best-effort scheduler	0.026%	0.004	0.035%	0.004

Table 1. Synthetic workload.

on. This behavior models an image server that attempts to exploit reordering of object delivery.

Table 1 shows the results of the two experiments, each repeated five times. In a lightly loaded system, the R-Scheduler is able to deliver almost all (99.99%) of the granted reservations within $\pm 50ms$. In the heavily loaded system, 99.95% are within this range. Since the R-Scheduler's resolution is $50ms$, these are good results. For both experiments, the overhead incurred by the admission controller (including the linear programming) as well as the R-Scheduler is well below 1% of the total experiment duration and thus close to negligible. The overhead of the R-Scheduler (written in C) is of the same order as the overhead by the built-in best effort scheduler (coded in assembler).

Both experiments were repeated with a R-Scheduler invocation period of $10ms$ and yielded similar percentages of requests granted within a $\pm 10ms$ window.

Instead of submitting a whole vector, an application could also repeatedly submit single requests until one of them is granted. Measurements have shown that, while the overhead of preparing a whole vector compared to a single request is not measurable, the vector interface performs better by a factor of two in case the 10^{th} request in the vector is granted, and by a factor of three for the 20^{th} request.

5.2. Real-life workload

Evaluations with the system-aware image retrieval system have shown that in a number of scenarios (one server, single request; multiple servers with same requests; and servers with multiple random requests) the R-Scheduler is very well able to shield applications from the detrimental effect of various background load. Furthermore, the image retrieval system with the R-Scheduler delivers a noticeably larger percentage of requests on time than when operating with the best-effort scheduler [6].

6. Related work

This section reviews some work from the area of CPU scheduling and compares it with the R-scheduler.

CPU reservations are central to real-time systems that schedule a fixed set of periodic, independent, non-blocking tasks with known, constant execution times [12]. Subsequent work relaxed those assumptions to allow for either aperiodic tasks [21] or for dependencies between tasks [20]. Neither the periodic nor the aperiodic approach to real-time scheduling can completely fulfill the needs of adaptive applications: *i*) it is questionable to map aperiodic tasks to the periodic model; *ii*) in both periodic and aperiodic cases it is possible to devise cases where one blocking task (i.e., waiting for I/O) causes another task to miss its deadline; *iii*) due to the dynamic runtime behavior of adaptive applications, it is impossible to calculate a feasible schedule off-line; and *iv*) pure real-time schemes are not designed to handle best-effort processes. The last item becomes obsolete with a combined best-effort/real-time scheduling system, but even with such a scheme the other drawbacks remain.

The scheduling scheme of [12] has also been deployed in a more dynamic environment: Mercer introduced the *Processor Capacity Reserves* abstraction [15] for measuring and controlling processor usage of real-time and multimedia applications under a microkernel architecture [23]. One goal of the work was to provide the predictability of real-time systems while retaining the flexibility of a time-sharing system. While Mercer's and our goals are similar, we go several steps beyond his approach: First, since Processor Capacity Reserves are being scheduled using real-time scheduling, they have the aforementioned disadvantages as far as adaptive applications are concerned. Second, Reserves have no end time specified in advance, and might result in over-reservation. Third, Mercer's dynamic QoS-server [11] is specifically tailored to multimedia applications requiring them to specify different predefined adaptation policies in advance. We, however, allow applications to decide at runtime about adaptation decisions.

Waldspurger introduced *lottery* and *stride scheduling* as novel resource allocation mechanisms providing efficient, responsive control over the relative execution rates of computations [24]. Subsequent work has applied proportional share scheduling to real-time tasks [22] or extended it to hierarchical partitioning of CPU bandwidth among various application classes that use their own scheduler [8]. In contrast to the proportional share model, which provides applications with a certain share of the CPU corresponding to a user-specified weight, the R-scheduler offers absolute, time-bounded resource reservations which can be contracted with the system in advance. We furthermore gracefully handle the case of blocking processes, which are not dealt with in Waldspurger's work at all.

Nieh and Lam presented *SMART*, a scheduler for multimedia applications [17] motivated by the observation that multimedia applications cannot be scheduled reasonably well with the real-time extensions of a standard time-sharing Unix scheduler [16]. In case of overload, SMART degrades conventional applications, and selectively drops slices of real-time applications, and notifies the applications thereof. In contrast to Nieh, the R-scheduler offers time-bounded reservations and prevents overload situations before they occur due to admission control; the negotiation and adaptation process takes place *before* the resources are used.

Jones et al. combined CPU reservations ("reserve X units of time out of every Y units") and time constraints (task must be run to completion between start time and its deadline) in the Rialto System [9]. From the conceptual point of view, time constraints are similar to our reservations. However, their scheduler's main domain are periodic applications. Additionally, a precomputed scheduling graph is used which does not allow for any flexibility regarding work-ahead or catch up.

Baker-Harvey presented a multiple-choice resource management scheme targeted primarily at periodic real-time applications [1]. Applications can submit a list of resource requests and associated functions that are called every period. Depending on the overall system load and a global policy, the Resource Manager decides for each application which of its resource requests can be satisfied. In contrast to Baker-Harvey, our approach is targeted at aperiodic applications with reservations that coexist with traditional best-effort applications on one single system.

Conventional operating systems like UNIX [19] 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 [14]. There have been various approaches to augment the basic scheme with scheduling functionality for real-time processes [3, 10], but it has also been shown that even for a static set of processes with different computational requirements, it is very hard if not impossible to find the right assignment of jobs to scheduling classes (e.g., either time-sharing or real-time) and within each class of jobs to priorities (real-time) or *nice*-values (time-sharing) [16].

7. Conclusions

This paper presents a comprehensive approach to processor management for adaptive applications. It consists of *(i)* an interface for an application to negotiate its future resource needs with the system by submitting a *vector* of resources reservation requests $R(C, I)$ where I is an interval and C denotes how many cycles are requested within I ; *(ii)* an ad-

mission controller using the Simplex Linear Programming method; and (iii) a reactive scheduling mechanism based on objective functions for cumulative resource consumption which enforces the granted reservations.

Experiments with a network-aware applications under varying usage scenarios and background loads provide strong evidence that the R-Scheduler allows adaptive applications to perform considerably better than with traditional best-effort scheduling and thus justifies the deployment of the R-Scheduler in operating systems.

As the importance of adaptive applications increases, operating systems are challenged to provide low-cost support for the adaptive CPU requests. The scheduler presented here – although using linear programming in the admission controller – provides an approach that can be easily integrated into existing systems and can co-exist with current best-effort scheduling disciplines.

References

- [1] M. Baker-Harvey. ETI resource distributor: Guaranteed resource allocation and scheduling in multimedia systems. In *Proc. of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 131–144, Feb. 1999.
- [2] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: A new facility for resource management in server systems. In *Proc. of the 3rd Symposium on Operating System Design and Implementation (OSDI)*, pages 45–58, Feb. 1999.
- [3] J. M. Barton and N. Bitar. A scalable multi-discipline, multiprocessor scheduling framework for IRIX. In *Proc. of Workshop on Job Scheduling Strategies for Parallel Processing*, Springer LNCS, Vol. 949, pages 45–69, Apr. 1995.
- [4] J. Bolliger and T. Gross. A framework-based approach to the development of network-aware applications. *IEEE Transactions on Software Engineering*, 24(5):376–390, May 1998.
- [5] J. Bolliger, T. Gross, and U. Hengartner. Bandwidth modelling for network-aware applications. In *Proc. of IEEE INFOCOMM 1999*, pages 1300–1309, Mar. 1999.
- [6] H. Domjan and T. Gross. Extending a best-effort operating system to provide QoS processor management. In *Proc. of the 9th International Workshop on Quality of Service*, 2001.
- [7] A. Fox, S. D. Gribble, E. A. Brewer, and E. Amir. Adapting to network and client variability via on-demand dynamic distillation. In *7th Intl Conference on Architectural Support for Programming Languages and Operating Systems*, pages 160–173, Cambridge, MA, October 1996.
- [8] P. G. X. Guo and H. M. Vin. A hierarchical CPU scheduler for multimedia operating systems. In *Proc. of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 107–121, Oct. 1996.
- [9] M. B. Jones, D. Roşu, and M.-C. Roşu. CPU reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Proc. of the 16th ACM Symposium on Operating System Principles*, pages 198–211, Oct. 1997.
- [10] S. Khanna, M. Sebree, and J. Zolnowsky. Realtime scheduling in SunOS 5.0. In *Proc. of the Winter 1992 USENIX Conference*, pages 375–390. USENIX Association, Jan. 1991.
- [11] C. Lee, R. Rajkumar, and C. Mercer. Experiences with processor-reservation and dynamic QOS in real-time mach. In *Proc. of Multimedia Japan, March 1996*, Mar. 1996.
- [12] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. of the Association of Computing Machinery*, 20(1):46–61, Jan. 1973.
- [13] D. Locke. *Best-Effort Decision Making for Real-Time Scheduling*. PhD thesis, Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, May 1986.
- [14] 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.
- [15] 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.
- [16] J. Nieh, J. G. Hanko, J. D. Northcutt, and G. A. Wall. SVR4 UNIX scheduler unacceptable for multimedia applications. In *Proc. of the Fourth International Workshop on Network and Operating Systems Support for Digital Audio and Video*, pages 35–48, Nov. 1993.
- [17] J. Nieh and M. Lam. The design, implementation and evaluation of SMART: A scheduler for multimedia applications. In *Proc. of the 16th ACM Symposium on Operating System Principles*, pages 184–197, Oct. 1997.
- [18] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C*. Cambridge University Press, 1991.
- [19] D. M. Ritchie and K. Thompson. The UNIX time-sharing system. *Comm. of the ACM*, 17(7):365–375, July 1974.
- [20] 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.
- [21] 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.
- [22] 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.
- [23] H. Tokuda, T. Nakajima, and P. Rao. Real-time mach: Towards a predictable real-time system. In *Proceedings of the USENIX Mach Workshop*, pages 73–82, Oct. 1990.
- [24] C. A. Waldspurger. *Lottery and Stride Scheduling: Flexible Proportional-Share Resource Management*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Sept. 1995.
- [25] R. Weber, J. Bolliger, T. Gross, and H.-J. Schek. Architecture of a networked image search and retrieval system. In *Proc. of the ACM Conference on Information and Knowledge Management (CIKM '99)*, pages 430–441, Nov. 1999.