Constant Bandwidth vs Proportional Share Resource Allocation

Luca Abeni, Giuseppe Lipari and Giorgio Buttazzo
Scuola Superiore S. Anna, Pisa
{luca,lipari,giorgio}@sssup.it

Abstract

Due to the success of new multimedia technologies, there is a great interest for providing a resource allocation strategy suitable for multimedia applications. Many solutions proposed in literature are based on proportional share resource allocation, such as WFQ, WF^{2}Q, EEVDF, or SFQ. In a previous paper, we showed how to perform resource reservation according to a Constant Bandwidth allocation. In this paper we compare Proportional Share and Constant Bandwidth resource allocation strategies, by showing advantages and drawbacks of both techniques.

1. Introduction

There exists a large class of applications that require real-time computation in order to be effective. An interesting example is given by multimedia systems, which manage Continuous Media (audio and video streams), for implementing video conference, telepresence, video on demand and other similar services.

In these systems, one of the main problems is to allocate the computational resources to the application tasks in order to provide some form of real-time execution. In fact, if resources are allocated independently from the temporal requirement of the application, the system may experience a degradation in Quality of Service (QoS), especially under high load conditions.

In the last years, the problem of modifying a conventional operating system to obtain fair resource allocation has been investigated. In [4], the authors proposed a variant of the Solaris, based on the SFQ scheduling algorithm. In [9], the Free-BSD scheduler has been modified to implement the EEVDF scheduling algorithm. In [10] the Linux kernel has been modified to provide proportional share. All these scheduling policies are essentially based on a common approach referred as Proportional Share resource allocation (PSA). The essence of PSA is to allocate a resource to a task for an uniform fraction of any time interval.

A different approach to the problem of providing QoS in an operating system is to adapt the real-time theory to a more flexible environment. In [6], a fixed priority scheme (derived from the Rate Monotonic algorithm) is used to realize an abstraction called Resource Reservation, which was implemented in Real-Time Mach. In [1] we presented a Constant Bandwidth resource allocation (CBA) strategy that, like PSA, lies between time sharing and real-time approaches. The essence of CBA is to reserve a fraction of the CPU bandwidth to each task, ensuring that each task cannot *demand* more than the reserved bandwidth. Constant Bandwidth allows uniform execution of Continuous Media (CM) tasks, whose time requirements are not exactly known a priori, without jeopardize the schedulability of possible real-time tasks which may execute in the same system.

In this paper we present a comparative analysis of the PSA and CBA approaches and show that the CBA can be a valid alternative to quantum-based PSA and can be more appropriate for supporting some hybrid real-time multimedia applications.

2. Background on proportional share resource allocation

The objective of the PSA model is to emulate a fluid flow system on a discrete quantum-based allocation system. The ideal scheduling model is the Generalized Processor Sharing (GPS) presented in [5]. It can be thought as the limiting form of a Weighted Round Robin policy: each task τ_i is assigned a *weight* w_i, which determines the minimum bandwidth sharing of the task. Given a task set $\Gamma = \{\tau_1, \ldots, \tau_n\}$, if $e_i(t_1, t_2)$ is the execution time of task τ_i in an interval $[t_1, t_2]$, the n tasks share the resource as follows:

$$\forall \tau_i \text{ active in } [t_1, t_2], \frac{e_i(t_1, t_2)}{e_j(t_1, t_2)} \geq \frac{w_i}{w_j} \; j = 1, 2, ..., n \quad (1)$$

Equation 1 expresses that in a GPS system each active task τ_i executes at least with a *rate* equal to $F_i = \frac{w_i}{\sum_{\tau_j \in \Gamma} w_j}$. The actual execution rate varies with time and is defined as the *share* of task τ_i: $f_i(t) = \frac{w_i}{\sum_{\tau_j \in \Gamma_a(t)} w_j}$ where $\Gamma_a(t)$ is the set of tasks which are active at time t.

In a Proportional Share system, the resource is allocated in discrete time quanta of length Q. A task acquires the resource at the beginning of a time quantum and can release it either at the end of the quantum or before. This is done by dividing each task τ_i in requests q_i^k having maximum size Q. Since the allocation is discrete in time, this approach generates an allocation error with respect to the ideal GPS model. Given two active tasks τ_i and τ_j, the allocation error in the time interval $[t_1, t_2]$ can be defined as $\epsilon_{i,j} = \left| \frac{e_i(t_1,t_2)}{w_i} - \frac{e_j(t_1,t_2)}{w_j} \right|$.

Another way of measuring the allocation error of a task is given by the *lag*. In the ideal GPS system, in the interval $[t_1, t_2]$ τ_i executes for a time $\int_{t_1}^{t_2} f_i(t)dt$; in a real system this is impossible because of the allocation error. The difference between the ideal and the real schedule is the lag:

$$lag_i(t_1) = \int_{t_0}^{t_1} f_i(t)dt - e_i(t_0, t_1),$$

where t_0 is the activation time of task τ_i.

The goal of a Proportional Share algorithm is to reduce the allocation error experienced by tasks. To support some form of real-time execution it is important to guarantee that $lag_i(t)$ is bounded. In fact, if an upper bound for $lag_i(t)$ exists, the execution time accumulated by τ_i is

$$e_i(t_0, t) \geq \int_{t_0}^{t} f(t)dt - max_t\{lag_i(t)\}. \qquad (2)$$

The first known Proportional Share scheduling algorithm is **Weighted Fair Queuing (WFQ)** [3], which emulates the behavior of a GPS system using the concept of *virtual time*. The virtual time $v(t)$ is defined by increments as follows:

$$\begin{cases} v(0) &= 0 \\ dv(t) &= \frac{1}{\sum_{\tau_i \in \Gamma_a(t)} w_i} dt \end{cases}.$$

Each request q_i^k is assigned a virtual start time $S(q_i^k)$ and a virtual finish time $F(q_i^k)$ defined as follows:

$$\begin{aligned} S(q_i^k) &= \max\{v(r_{i,k}), F(q_i^{k-1})\} \\ F(q_i^k) &= S(q_i^k) + \frac{Q_{i,k}}{w_i} \end{aligned}$$

where $r_{i,k}$ is the time at which request q_i^k is generated and $Q_{i,k}$ is the request dimension (required execution time). Since $Q_{i,k}$ is not known a priori, it is assumed equal to the maximum value Q_i. Tasks' requests are scheduled in order of increasing virtual finish time: in the virtual time domain, each request will finish before its virtual finish time.

In static systems, where all the tasks are always active, WFQ provides fairness by bounding the allocation error. However, it presents the following problems: it needs a frequent recalculation of $v(t)$, increasing runtime overhead; it does not perform well in dynamic systems; it assumes each requests size equal the maximum value: in a real situation this assumption is not correct.

In [4], a proportional share scheduler is used to subdivide the CPU bandwidth between various application classes: the proposed algorithm, **Start Fair Queuing (SFQ)**, is similar to WFQ but defines the virtual time in a different manner and schedules the requests in order of increasing virtual start time. The virtual time $v(t)$ is defined as follows:

$$v(t) = \begin{cases} 0 & \text{if } t = 0 \\ 0 \text{ or any value} & \text{if the CPU is idle} \\ S(q_i^k) & \text{if request } q_i^k \text{ is executing} \end{cases}$$

SFQ calculates $v(t)$ in a simpler way (introducing less overhead) and does not need the virtual finish time of a request to schedule it, so it does not require any a priori knowledge on the request execution time ($F(q_i^k)$ can be calculated at the end of q_i^k execution).

SFQ provides a lag bound $\max_t\{lag_i(t)\} = Q_i + F_i \sum Q_j$ which depends on the number of active tasks.

In [9], the authors propose a scheduling algorithm, called Earliest **Eligible Virtual Deadline First (EEVDF)**, that provides a bound on the lag experienced by each task. EEVDF defines the virtual time as WFQ and schedules the requests by virtual finish times (in this case called virtual deadlines), but uses the virtual start time (called virtual eligible time) to decide whether a task can be scheduled (i.e. is eligible): if the virtual eligible time is greater than the actual virtual time, the request is not eligible. Virtual start and finish times are defined as follows:

$$\begin{aligned} S(q_i^k) &= \max\{v(r_{i,k}), S(q_i^{k-1}) + \frac{Q_{i,k-1}}{w_i}\} \\ F(q_i^k) &= S(q_i^k) + \frac{Q_{i,k}}{w_i}. \end{aligned}$$

When a task activates or blocks, $v(t)$ is adjusted in order to maintain the fairness in dynamic systems. The lag bound guaranteed by EEVDF algorithm is Q, which is the minimum lag bound, for this reason, EEVDF is said to be optimal. EEVDF can also schedule dynamic task sets and can use fractional and non uniform quantum size, so it can be used in a real operating system.

3. Background on constant bandwidth resource allocation

Dynamic real-time systems try to guarantee timing requirements by reserving each task all the resources it needs for its execution. If such a guarantee cannot be done, the task is rejected. To perform admission control, all the resources used by a task have to be known a priori and the temporal requirements have to be explicitly declared.

A task τ_i is viewed as a stream of jobs $J_{i,j}$, each of them characterized by an arrival time $r_{i,j}$ and an absolute deadline $d_{i,j}$, meaning that the correct execution of the task requires that each job execution finishes within its deadline. Once the admission test is passed, tasks can be scheduled by absolute deadlines (Earliest Deadline First, EDF) or by periods (Rate monotonic, RM).

Unfortunately, some of the task parameters (for example the jobs' execution times) cannot be exactly estimated before execution. For this reason the admission test (and the resource reservation) is performed based on *declared* values; if each task respects the declared requirements, each of its jobs will finish before its deadline, but if a task requires more resources than declared, the guarantee mechanism is perfectly useless.

A solution to the problem is given by the Resource Reservation [6] abstractions: a task is allowed to use a resource for C_i time units each T_i; after this time the reserve is depleted and will be recharged at the next period.

Constant Bandwidth allocation [1] uses a similar solution by introducing the concept of *demanded bandwidth*, calculated on absolute deadlines. We define the demanded bandwidth of task τ_i as

$$B_i = \max_{t_1, t_2 : t_2 > t_1} \frac{D_i(t_1, t_2)}{t_2 - t_1} \tag{3}$$

where $D_i(t_1, t_2)$ is the time demanded by τ_i in the interval $[t_1, t_2]$ (i.e., the sum of the execution times of jobs $J_{i,j}$ having release time $r_{i,j} \leq t_1$ and served with deadline $d_{i,j} \leq t_2$). Each task is scheduled such that its demanded bandwidth B_i never exceeds the reserved bandwidth U_i. Thus if $\sum_\Gamma U_i \leq 1$, it is guaranteed that it will receive at least the demanded bandwidth.

In [1], it is shown how it is possible to realize a Constant Bandwidth CPU allocation using a dedicated aperiodic server (the Constant Bandwidth Server, CBS) for each task.

Each aperiodic server with a bounded demanded bandwidth B_i can be used for realizing CBA, but CBS permits hard real-time guarantee if tasks' parameters are known. The CBS assigns each job an initial deadline, which is postponed each time the task is demanding more than the reserved bandwidth. The server is described by two parameters Q_s and T_s, where T_s is the *server period*, Q_s is the service time assigned to the task in each period, and $U_s = \frac{Q_s}{T_s}$ is the server demanded bandwidth. Deadlines are assigned to jobs as follows. When a new request arrives, the server checks whether the last assigned deadline can be used, otherwise it assigns to the request an initial deadline equal to $r_{i,j} + T_s$. Each time the job executes for Q_s time units, its scheduling deadline is postponed by T_s (a complete description of the algorithm can be found in [1]).

Using a CBS to implement Constant Bandwidth resource allocation, the following lemma can be proved:

Lemma 1 *If a task τ_i served by a CBS with parameters (Q_s, T_s) is active in $[t_0, t]$ and d is a scheduling deadline assigned by the CBS (with $t_0 \leq d \leq t$), then*

$$e_i(d, t) \geq \left\lfloor \frac{t - d}{T_h} \right\rfloor Q_h. \tag{4}$$

Equation 4 expresses the demanded time when a CBS is used.

In [7] Spuri and Buttazzo use a similar mechanism to schedule aperiodic tasks realizing the resource allocation with a Total Bandwidth Server. The problem with this server is that *it cannot be used when tasks' WCET are not known*.

The Resource Reservation abstraction [6] does not introduce the concept of reserved bandwidth, but it is similar to CBS in that it realizes a CBA. Like CBS, it allows to perform hard guarantee, but, being based on fixed priority, *it performs worse in overload situations*. In facts, when a task executes for C_i time units in its period T_i and does not complete (the reservation is depleted), the rest of the task is scheduled in background. Using dynamic priority, CBS schedules the depleted tasks by postponing deadlines.

4. Parallelism between PSA and CBA

In this section we compare the behavior of the Proportional Share and Constant Bandwidth approaches, in terms of objectives, task model and run time overhead.

Similarities:
Looking at the definitions given in Sections 2 and 3, it is easy to see how the demanded bandwidth B_i is similar to rate F_i. The main difference in this parameters is that by definition $\sum_\Gamma F_i = 1$, while $\sum_\Gamma B_i \leq 1$ (using the constant bandwidth abstraction, a fraction of the CPU bandwidth can be left unallocated).

In [8], the authors show how the real-time tasks' share can be maintained constant at the arrival of a new task (Γ varies). This is done by re-arranging the tasks' weights: like for CBA, there is an admission test $\sum_{\tau_i \in \Gamma} \frac{C_i}{T_i} \leq 1$ (similar to $\sum_{\tau_i \in \Gamma} U_i \leq 1$) and, if a new task is accepted, the weights are recomputed such that for all real-time tasks $F_i = f_i(t) = \frac{C_i}{T_i}$. This is not necessary under the constant bandwidth paradigm, where if a new task is accepted the other tasks' bandwidth have not to be changed.

This difference is due to the fact that F_i is computed based on the *executed* time, while B_i is computed based on the *demanded* time.

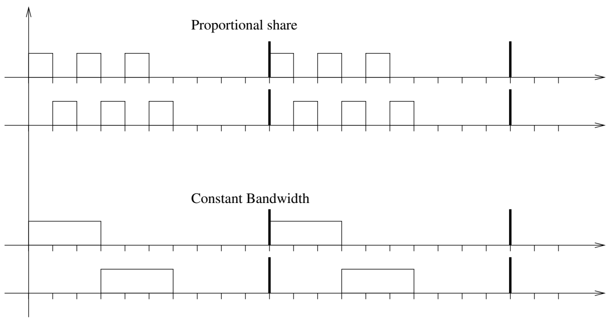

Figure 1. Examples of PSA and CBA

Moreover, with a CBS, the additional parameter T_s (period of the server) can be used to better describe the tasks' temporal behavior.

Differences:

There is a fundamental difference between the two theoretical models from which Proportional Share and Constant Bandwidth derive: comparing Equation (2) with Equation (4) we can see how a proportional share scheduler tries to maintain the ratio $\frac{e(t_0,t_1)}{t_1-t_0}$ constant over *any* time interval $[t_0, t_1]$, while a constant bandwidth scheduler maintains this ratio constant only *between deadlines*.

We believe that imposing a fair execution over any time interval overconstraints the system, since in most soft real-time or multimedia application it is important to schedule tasks before their deadlines, not to schedule them in a fair way. For example, in a video or audio player it is important to decode a frame before the start of the next frame, not to decode it fairly. A Proportional Share scheduler must schedule a task in this fair way to respect its QoS requirements because the scheduler does not know an important task's parameter like the *period* (or the *deadline*).

Another fundamental difference between the Proportional Share paradigm and the Constant Bandwidth paradigm is that the former allocates the resources in time quanta, while the latter does no require so (it is fully preemptive). This makes Proportional Share easier to implement in a conventional operating system (such as Unix or Windows), whose scheduler is quantum-based, but introduces some limitations: for example, tasks can be activated only at quantum boundaries and tasks' periods must be multiple of Q. Moreover, a quantum-based allocation algorithm causes an overhead (an interrupt is generated at each quantum boundary) and a larger allocation error.

The fairness property enforced by Proportional Share schedulers generates a larger number of preemptions, as shown in Figure 1, and the performance of the system can be penalized by the additional preemption overhead (in a real system, context switches could take some time!).

Another problem with the Proportional Share paradigm is that it does not model event-based systems intuitively. For example, a task activated by an external event (a task that manages data from the network, or that have to respond to an interrupt) can only be modeled as a periodic task, and it is not clear how to assign the weight or the share to this task. On the other hand, using CBA, we can reserve a given bandwidth to an aperiodic event-driven task and use the server parameter T_s to model the expected interarrival time. Using a CBS to serve the task, it is also simple to perform a deterministic or stochastic guarantee on the response time.

Duality between PSA and CBA:

In summary, we can say that Constant Bandwidth and Proportional Share give different interfaces to similar (but not equal) allocation models: the Constant Bandwidth programming model is more suited for handling real-time constraints, so the CBA is to be preferred for integrating multimedia streams in a real-time system, while PSA is closer to the classical time-sharing approach, hence a Proportional Share scheduler can be more easily integrated in a conventional operating system.

As shown, CBA provides explicit support for hard real-time execution (reserving a bandwidth $B_i = \frac{WCET_i}{T_i}$ to each hard task, or scheduling it directly with the EDF algorithm), but also PSA can emulate it, at the cost of the additional overhead of dynamically rearranging the tasks' weights.

On the other hand, Proportional Share schedulers are more flexible in partitioning the bandwidth among non-guaranteed tasks: this is useful to run non real-time applications (such as in a traditional workstation) together with multimedia ones. In this case, the notion of weight is more intuitive than the reserved bandwidth; Constant Bandwidth can emulate this, by defining $B_i = F_i$, $T_i = Q/B_i$ and using non Real-Time (NRT) tasks (an NRT task is a task composed by a single, finite or infinite, job).

5. Simulation results

To compare the performance of the proposed methods, we performed some simulations on an hybrid task set, composed of soft and hard tasks, using the scheduling simulator described in [2].

The task set is composed of 5 periodic hard tasks (fixed period, fixed execution time), which generate a hard load $U_{Hard} = \sum_{i=1}^{5} \frac{C_i}{T_i} = 0.35$, and 3 soft tasks, whose interarrival and execution times are uniformly distributed around the mean values $\overline{t_{Soft}}$ and $\overline{c_{Soft}}$, being $U_{Soft} = 3\frac{\overline{c_{Soft}}}{\overline{t_{Soft}}}$.

The performance of the algorithms has been evaluated measuring the mean *Tardiness*: the tardiness of a job is defined as 0 if the job finishes before $\overline{t_{Soft}}$, and the difference between the finishing time and $\overline{t_{Soft}}$ divided by $\overline{t_{Soft}}$ otherwise.

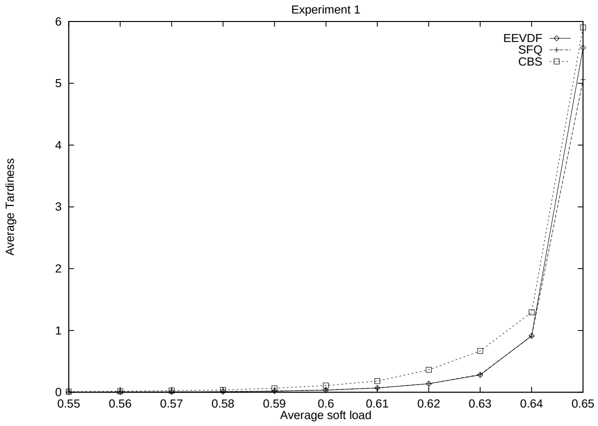

Figure 2. Mean tardiness experienced by CB and PS schedulers

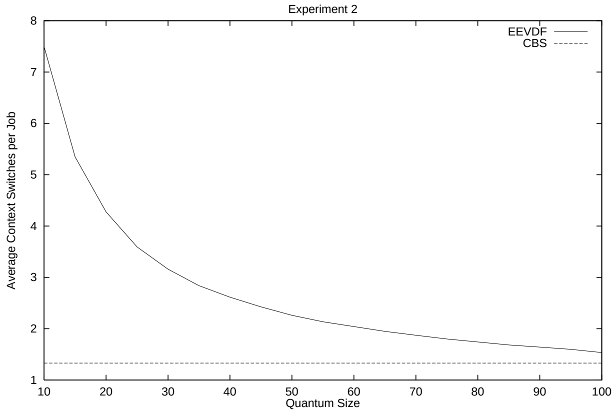

Figure 3. Mean number of context switch per job

In the first experiment (Figure 2), we compare SFQ and EEVDF (using a quantum size of 1) against CBS. Using CBS, each hard task is scheduled by EDF, while soft tasks are scheduled by a dedicated server with parameters $(\overline{c_{Soft}}, \overline{t_{Soft}})$; using a PS scheduler the weights are assigned so that no hard deadlines are missed, whereas all soft tasks have the same weight.

Looking at Figure 2 it is easy to see that EEVDF and SFQ perform slightly better than CBS (because they are fairer), but this is due to the fact that we are using a very small quantum size. Although small quantum permits to reduce the mean tardiness, this causes a big number of context changes. To see this, Figure 3 shows the mean number of context switches experienced by a single task instance as a function of the quantum size, when a PS scheduler is used. Even using a quite small quantum (for example 10), the number of context switches enforced by the PS scheduler is much greater than the CBS (between 5 and 6 times). On the other hand, if the quantum size is increased to limit the number of context switches, the performance of a PS scheduler becomes closer to the CBS one.

Further increasing the scheduling quantum size causes a waste of CPU bandwidth, so it is impossible to guarantee hard tasks using a proportional share scheduler with a too big quantum size.

6. Acknowledgements

This work has been carried out under the financial support of the Ministero dell'Università e della Ricerca Scientifica e Tecnologica (MURST) in the framework of the Project "Design Methodologies and Tools of High Performance Systems for Distributed Applications".

References

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

[2] A. Casile, G. Buttazzo, G. Lamastra, and G. Lipari. Simulation and tracing of hybrid task sets on distributed systems. In *Real Time Computing Systems and Applications*, 1998.

[3] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing alorithm. In *ACM SIGCOMM*, pages 1–12, September 1989.

[4] P. Goyal, X. Guo, and H. M. Vin. A hierarchical cpu scheduler for multimedia operating systems. In *2nd OSDI Symposium*, October 1996.

[5] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking*, 1(3):344–357, June 1993.

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

[7] M. Spuri and G. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10(2), 1996.

[8] I. Stoica, H. Abdel-Wahab, , and K. Jeffay. On the duality between resource reservation and proportional share resource allocation. In *SPIE Multimedia Computing and Networking*, volume 3020, pages 207–214, San Jose, CA, February 1997.

[9] 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 *IEEE Real Time System Symposium*, December 1996.

[10] C. A. Waldspurger and W. E. Weihl. Stride scheduling: Deterministic proportional-share resource mangement. Technical Report MIT/LCS/TM-528, Massachusetts Institute of Technology, June 1995.