Resource Management on Multi-core Systems: the ACTORS approach*

Enrico Bini, **Giorgio Buttazzo**
Scuola Superiore Sant'Anna, Pisa, Italy

Johan Eker
Ericsson Research, Lund, Sweden

Stefan Schorr, **Raphael Guerra**, **Gerhard Fohler**
Technische Universität Kaiserslautern, Germany

Karl-Erik Årzén, **Vanessa Romero**
Lund University, Lund, Sweden

Claudio Scordino
Evidence Srl, Pisa, Italy

December 23, 2010

Abstract

High-performance embedded systems, such as smart phones, require the execution of many applications on multi-core platforms. Theses systems are subject to a number of stringent restrictions and constraints, such as limited amount of CPU power, long battery life, i.e., low energy demands, but also timeliness, e.g., for call handling or video playback. Key aspects for achieving this goal are: isolation between applications and assignment of appropriate amounts of computational resources to each application.

The approach developed in the ACTORS project provides temporal isolation through resource reservation over a multi-core platform, adapting the available resources based on the overall quality requirements. The architecture is fully operational on both ARM MPCORE and the x86 multi-core platforms.

1 Introduction

Although smart phones and other consumer products are de facto full blown computers, they are commonly not regarded as general purpose devices. Full programmability is sacrificed for ease of use and portability. To some extent the main differences between a modern cell phone and a computer concern interface and user expectations.

When a home computer fails, we do not get surprised, we get annoyed, but when our cell phone crashes we return it to the store. Consumer devices are expected to be always on and always working. Today, advanced cell phones show a strong UNIX heritage, e.g. support for processes and threads, separate memory spaces, etc., but none of these features are exposed to the user. Most features are even hidden to the application programmers. This is particularly true for some operating system (OS) mechanisms, like task scheduling. One of the main reasons why multitasking in cell phone OS's is highly restricted for 3rd party developers is the problem of resource management, i.e. which tasks and applications gets to consume CPU cycles, battery life time, communication bandwidth, etc. Cell phone manufacturers carefully tune the resource distribution and scheduling to optimize user experience and then lock the API's. Multitasking is commonly limited to a number of APIs that provide services (e.g. audio streaming) running in the background. A fundamental problem is that most standard operating systems only provide very crude mechanisms to distribute resources between ap-

*This work has been partially supported by the European Commission under the ACTORS project (FP7-ICT-216586).

plications, mainly using priorities. The priority assignment, however, prevents the isolation in multi-application systems, since it requires a system-wide knowledge of all applications and services competing for the same resources.

We believe that resource reservations [10, 1] provide a more suitable interface for allocating resources such as CPU to a number of applications. According to this method, each application is assigned a fraction of the platform capacity, and it runs as if it were executing alone on a less performing *virtual platform*, independently of the behavior of the other applications. In this sense, the temporal behavior of each application is not affected by the others and can be analyzed in isolation.

The concept of virtual platform is not new. Nesbit et al. [12] introduced the *virtual private machine* that provides an abstract view of all the physical resources (processors, memory bandwidth, caches, etc.) along both spatial and temporal dimension. The abstraction for virtual platforms proposed in this project, instead, provides a finer control of the CPU time that allows guaranteeing applications with real-time requirements. Indeed the abstraction of the only CPU time introduces a degree of uncertainty due to the lack of modeling of other kind of resources. In ACTORS we manage this uncertainty with a feedback loop that adjusts the CPU time also according to the final QoS delivered to the user.

The ideas presented is this paper were driven by the wish to automatize the distribution of available resources, not only at design time, but also at run time, based on the actual user demand. For operating mechanical and electrical systems in uncertain environments, feedback control is a key methodology. We show that the same approach can be adopted for controlling resource usage in a software application. The amount of computational resources allocated to an application can be effectively used as an actuator to apply feedback control. Defining sensors is in general more complex and application dependent. In the ACTORS project[1], we focus on streaming applications, e.g. audio and video codecs, radio receivers and transmitters, etc., which have an inherent notion of progress, in that the workload is divided into a number of frames and there exists a nominal, expected rate, e.g., frames per second. Although the framework presented in this paper is not limited to streaming applications, we believe it is particularly suited for the large class of devices performing real-time tasks, such as video decoding and streaming.

Multi-Core systems add a new dimension to the resource management problem. In fact, the actual load has to be partitioned onto the available computational resources, which may depend on specific objectives, such as maximizing performance, limiting the peak temperature of the chip, or maximizing battery life time. Depending on the specific hardware, there can be possibilities to shut down cores or perform system wide or individual voltage scaling to manage energy.

To respond to all such requirements, the ACTORS research project proposed the software architecture depicted in Figure 1.

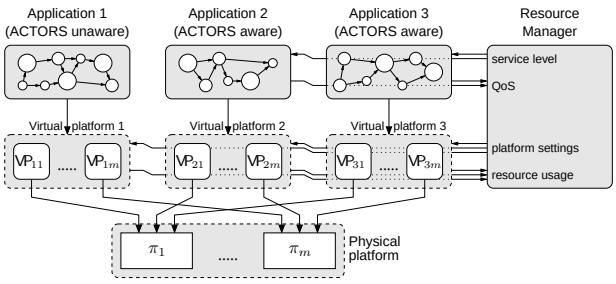

Figure 1: The ACTORS architecture.

To perform resource allocation independently of a particular physical platform, applications are allocated to virtual platforms. A virtual platform consists of a set of *virtual processors* (VPs), each executing a portion of an application. Each virtual processor is implemented as a Linux reservation server that is allocated onto a physical core.

To cope with variability, or when the computational requirements are not precisely specified, applications notify the achieved *Quality of Service* (QoS) to a Resource Manager (RM). The RM adapts the reservations on the virtual processors using a feedback loop that measures the resource actually con-

[1] `http://www.actors-project.eu/`

sumed, and notifies the applications about the newly assigned service levels. The system also allows executing applications that cannot specify their QoS levels (ACTORS unaware applications).

2 Resource Abstraction

A key design requirement for simplification of portability between platforms is application development independent of physical platforms. This is crucial for multi-core systems where adding a processor may not necessarily bring an improvement of performance [6]. In fact, embedded software developed to be highly efficient on a given multi-core platform, could be very inefficient on a new platform with a different number of cores.

Roughly speaking, two key features have to be extracted to properly virtualize a multi-core platform: the overall computing power α of the entire multiprocessor platform and the number m of virtual processors of the platform. While a fully parallelizable application is only affected by α (since it can exploit any degree of parallelism), a fully serial application can only execute on one core, hence for the same overall capacity α, the smaller m the better.

In ACTORS, a virtual multi-core platform is represented as a set of m *sequential* virtual processors (VPs). We briefly recall the basic concepts that allow the abstraction of a single processor.

2.1 Virtual processor abstraction

The abstraction of a virtual processor should provide a measure of the amount of computation the VP can provide. For this purpose, the concept of *supply function* has been introduced [11, 8].

> The supply function $Z_k(t)$ of a virtual processor VP_k represents the minimum amount of resource that VP_k can provide in any time interval of duration t.

A supply function is non-negative, monotonic, and super-additive. For example, a periodic server that provides a budget of Q units of time every period P has a supply function as the one reported in Figure 2. Unfortunately, abstracting a VP by the pair (Q, P) is inappropriate if the actual implementation mechanism is not a periodic server, but a static partition of time. Moreover, at an early stage of the design process, it is recommended that an abstraction does not rely on a specific implementation mechanism, such as the pair (Q, P).

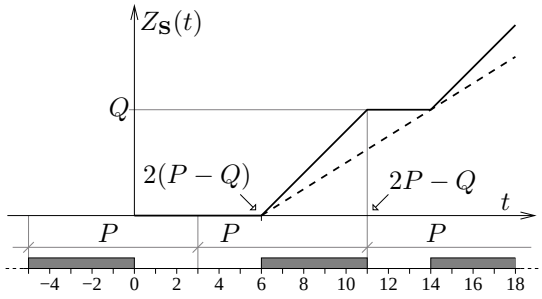

Figure 2: Supply function of a periodic server.

In ACTORS a virtual processor is described by the *bounded delay abstraction* introduced by Mok et al. [11] to represent a virtual platform on a wide class of possible implementations: a VP is described by means of two parameters: a bandwidth α and a delay Δ. The bandwidth α measures the amount of resource assigned to the demanding application, whereas Δ represents the worst-case service delay. This abstraction has also the additional benefit of being common to other fields, such as networking [14] and disk scheduling [4]. Thus, the analysis proposed here can also be extended to a more complex system including different types of resources.

Informally speaking, the bandwidth α and the delay Δ describe the tightest linear lower bound to the supply function. In Figure 2, such a linear lower bound is represented by a dashed line. Given the supply function $Z_k(t)$ of the VP_k, the two parameters are formally defined as

$$\alpha_k = \lim_{t\to\infty} \frac{Z_k(t)}{t} \qquad \Delta_k = \sup_{t\geq 0} \left\{ t - \frac{Z_k(t)}{\alpha_k} \right\}.$$

Indeed, the bandwidth captures the most significant feature of a VP. However, two VPs with the

same bandwidth can allocate time in a significantly different manner: suppose that a VP allocates the processor for one millisecond every 10 msec and another one allocates the processor for one second every 10 seconds. Both VPs have the same bandwidth (10% of the physical processor), however, the first VP is more *responsive* in the sense that an application can progress more uniformly. The Δ parameter provides a measure of the responsiveness, as proposed by Mok et al. [11]. For example, a periodic server with period P and budget Q has the following α, Δ parameters:

$$\alpha = \frac{Q}{P} \qquad \Delta = 2(P - Q) \tag{1}$$

as also illustrated in Figure 2.

2.2 Virtual multiprocessor abstraction

The abstraction of a virtual multiprocessor must also represent the degree of parallelism. Hence, a virtual multiprocessor is represented by a set of virtual processors [3], formally modeled by the vector of m pairs $[(\alpha_1, \Delta_1), (\alpha_2, \Delta_2), \ldots, (\alpha_m, \Delta_m)]$. Real-time constraints can be guaranteed on top of this abstraction [3]. A parallel application is then partitioned into subsets, each mapped onto a single sequential VP.

3 Linux Implementation

Linux was initially designed to be a general purpose OS for servers and desktop environments, thus not much attention has been dedicated to real-time issues. In particular, the current scheduling framework (recently introduced by Ingo Molnar as a substitute for the previous $\mathcal{O}(1)$ scheduler) does not contain any resource reservation mechanism capable of guaranteeing real-time constraints. This section explains how the Linux scheduling framework (kernel release 2.6.33) has been extended to include a resource reservation scheduler.

The Linux scheduling framework contains an extensible set of scheduling modules, called *scheduling classes*. Each class implements a specific scheduling algorithm and schedules tasks having only a specific policy. At run-time, the scheduler core inspects the policy of the running tasks and schedules each task selecting the proper algorithm. A priority order among the scheduling classes ensures that low-priority tasks are not executed when higher-priority tasks are ready. In the considered kernel release, only two scheduling classes are available (see Figure 3):

- `sched_fair` implements the *"Completely Fair Scheduler"* (CFS) algorithm, and schedules tasks having `SCHED_OTHER` or `SCHED_BATCH` policies. Tasks are run at precise weighted speeds, so that each task receives a "fair" amount of processor share.
- `sched_rt` implements a POSIX fixed-priority real-time scheduler, and handles tasks having `SCHED_FIFO` or `SCHED_RR` policies.

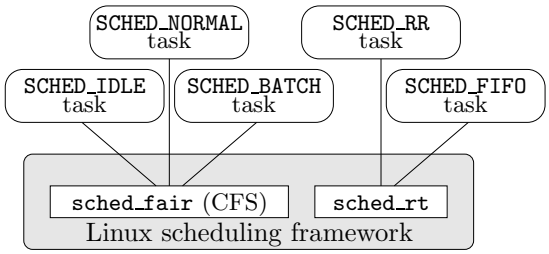

Figure 3: Linux scheduling framework.

Unfortunately, none of these scheduling classes are suitable for reserving a share of the processor to tasks with timing constraints [5]. The API available on Linux, in fact, allows assigning a share of processor time to a task, but it does not allow associating temporal constraints (e.g., deadlines) to this share.

Therefore, the existing scheduling algorithms cannot ensure that the tasks will receive the specified amount of CPU cycles before a given time. Even worse, using CFS, the time slice assigned to a task is not deterministic and cannot be known in advance, since it depends on the number of tasks running in the system at that time.

Usually, this problem is solved by over-allocating the amount of CPU to time-sensitive tasks, so that

they will receive the CPU more often, meeting their timing constraints. Such a strategy, however, is counterproductive, especially in the field of embedded systems, where the need of reducing the amount of resources used by the system is critical. Moreover, this strategy cannot guarantee that tasks' timing constraints will be met in *all* circumstances.

3.1 The Linux Deadline Scheduler

Within the ACTORS project, the new scheduling class `SCHED_DEADLINE`, based on resource reservations, has been implemented and proposed to the kernel community[2]. The class adds the possibility of scheduling tasks using the Constant Bandwidth Server (CBS) [1], without changing the behavior of tasks scheduled using the existing policies. CBS, in turn, is based on Earliest Deadline First (EDF) task queues. The implementation does not make restrictive assumptions on the characteristics of tasks.

During the last years, several real-time schedulers have been proposed for Linux. However, none of them eventually became part of the official Linux kernel. Our approach differs from the existing schedulers in the following aspects:

- It is integrated with the mainline Linux scheduler.
- It is a new, self-contained scheduling class instead of a kernel module invoked through *"hooks"* in the main scheduling class.
- It natively supports multi-core platforms.
- It is not architecture-dependent, therefore it can run on all platforms already supported by Linux.
- It is a hierarchical scheduler, where tasks can be grouped in Virtual Processors characterized by budgets and periods. The hierarchy is handled through the standard Linux mechanisms (i.e., Control Groups).

Extensive experiments [9] showed that tasks executed under `SCHED_DEADLINE` instead of `SCHED_FIFO` can better meet their timing requirements. For example, when `SCHED_FIFO` is used to schedule two instances of a GTK video player, only one instance of the player can meet its timing constraints (i.e., 25 frames per second). When `SCHED_DEADLINE` is used, instead, both instances can reproduce the movie smoothly, and the performance of each instance is proportional to the share of CPU assigned through the real-time parameters.

[2]The code is freely available on the public Git repository `http://gitorious.org/sched_deadline`.

3.2 User-level interface

The `SCHED_DEADLINE` class has been modified into the `SCHED_EDF` class to meet the specific needs of ACTORS. The new system call `sched_setscheduler2` allows creating or modifying task's budget and period. These values must be provided for non-periodic tasks as well, so that they are constrained to execute no longer than their budget every period, ensuring a proper isolation between running tasks.

For periodic tasks, the semantics of the `sched_yield` system call has been extended to inform the kernel that the current task instance has finished execution. In this case, the current task is blocked until the end of the current period. This system call must be invoked by the task itself at the end of each period. If the task belongs to a Virtual Processor, the other tasks of the VP can continue their execution.

Virtual Processors can be created and destroyed using the same cgroups interface available for the other scheduling classes — i.e., through the `mount` Linux command. The cgroups interface also allows binding tasks to a Virtual Processor and setting or changing its reservation parameters. Budget and period of a VP can be set using two entries in the cgroups filesystem (`cpu.edf_runtime_us` and `cpu.edf_period_us`, respectively). These files are created once the cgroup filesystem is mounted.

The system maintains a hierarchy of Virtual Processors. For this reason, without setting budget and period of the "root" Virtual Processor, the other Virtual Processors cannot receive any amount of CPU time.

Virtual Processors export a further entry in the cgroup interface, called `edf_reservation_data`.

This file contains information about the run-time behavior of the group of tasks. In particular, it contains:

- the amount of budget used by the Virtual Processor so far;
- the number of times the budget of the Virtual Processors has been recharged;
- the number of times the Virtual Processor entered the *Hard Reservation* state, that is the VP has been blocked because it finished the whole budget available in a period.

4 Resource Manager

In ACTORS, decisions about allocation of resources to application are performed by a Resource Manager (RM), which is implemented as user-level application. Frequent reactions to fluctuations of computational demands and resource availability would be too inefficient. Rather, resource management in ACTORS is inspired by the MATRIX resource management framework [13] where application demands are abstracted as a small set of *service levels*, each characterized by a QoS and resource requirements. In this way, only significant changes trigger a system reconfiguration.

4.1 Service Level Assignment

Table 1 shows an example of an application that can operate at four service levels using four VPs. In the

SL	QoS [%]	α [%]	Δ [ms]	BWD [%]
0	100	200	50	$[50, 50, 50, 50]$
1	80	144	90	$[36, 36, 36, 36]$
2	50	112	120	$[28, 28, 28, 28]$
3	30	64	250	$[16, 16, 16, 16]$

Table 1: Service level table of an application.

table, SL, QoS, α, Δ, and BWD denote the service level index, the quality of service, the total bandwidth, the tolerable application delay, and the bandwidth distribution over the individual virtual processors, respectively. The total bandwidth requirement α is expressed as a percentage of the computational power of a single core. In the example above, the total bandwidth is evenly distributed over the multicore platform. These values are only initial estimates provided by profiling tools and are tuned during run time by a feedback loop.

During the initialization phase, applications register with the RM to announce their available service levels. After a successful registration, the RM selects the appropriate service level for each application using an integer linear programming (ILP) solver, whose objective is to maximize the global QoS of the system, under the constraint that the total amount of resources is limited.

Once service levels are assigned, the RM distributes the bandwidth. VPs are created through the Linux interface described in Section 3. The budget and the period of the server are computed based on the corresponding (α, Δ) parameters, as described in Equation (1). The mapping of VPs is also formulated as an ILP problem.

4.2 Resource Adaptation

Resource adaptation is achieved through a control mechanism, which uses a combination of feedforward and feedback strategies to modify resource allocation at run time. The service level assignment and the bandwidth distribution constitute the feedforward part of the resource allocation strategy. The feedback part consists of one bandwidth controller for each VP. The bandwidth controller checks whether the tasks within the VP are making optimal use of the allocated bandwidth, and takes actions to avoid wasting resources without degrading the performance of the application. Bandwidth controllers are executed periodically with a period that is a multiple of the period of the VP they are controlling.

The bandwidth controllers measure the actual resource consumption using two measurements provided by the Linux interface:

1. the *used bandwidth* (UB), which is the average used budget over the sampling period of the controller, and

2. the *exhaustion percentage* (EP), which is the percentage of server periods over the last sampling interval in which the VP budget is totally consumed. This indicates that the application was likely to require more bandwidth.

The bandwidth controllers have a cascade structure shown in Figure 4. The controller affects the application behavior through the Linux interface. $\mathsf{EP_s}$ denotes the maximum percentage of budget exhaustion that can be tolerated by the application. The controller C_1 defines the new values of the set point for the used bandwidth $\mathsf{UB_s}$. The controller C_2 adjusts the VP bandwidth by modifying the assigned bandwidth AB.

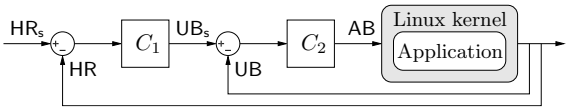

Figure 4: Structure of the bandwidth controller.

The feedforward/feedback structure employed in ACTORS is based on the AQuoSA architecture [2], employed in the EC FRESCOR project. The main extensions developed in ACTORS are the support for multi-core applications and the support for hard rather than soft reservations.

5 Experimental evaluation

This section presents two simulation experiments to illustrate the capability of the bandwidth adaptation mechanism to cope with variable loads.

The first experiment is carried out using a simple application, called *periodic pipeline*, performing its computation in four different stages. Each stage executes on a dedicated VP. Each of the four VP ($\mathsf{VP_1}$, $\mathsf{VP_2}$, $\mathsf{VP_3}$, and $\mathsf{VP_4}$) is bound over a different physical processor. The first stage is activated periodically. The next stages are activated upon the completion of the preceding ones. The application has four service levels reported in Table 1.

Figure 5 shows the evolution over time of the used bandwidth (UB), the assigned bandwidth (AB), and the exhaustion percentage (EP). The x-axis reports the simulation time in seconds. At time 0, the periodic pipeline application registers with the RM, which assigns the highest service level (0) since it is the only running application. The initial bandwidth

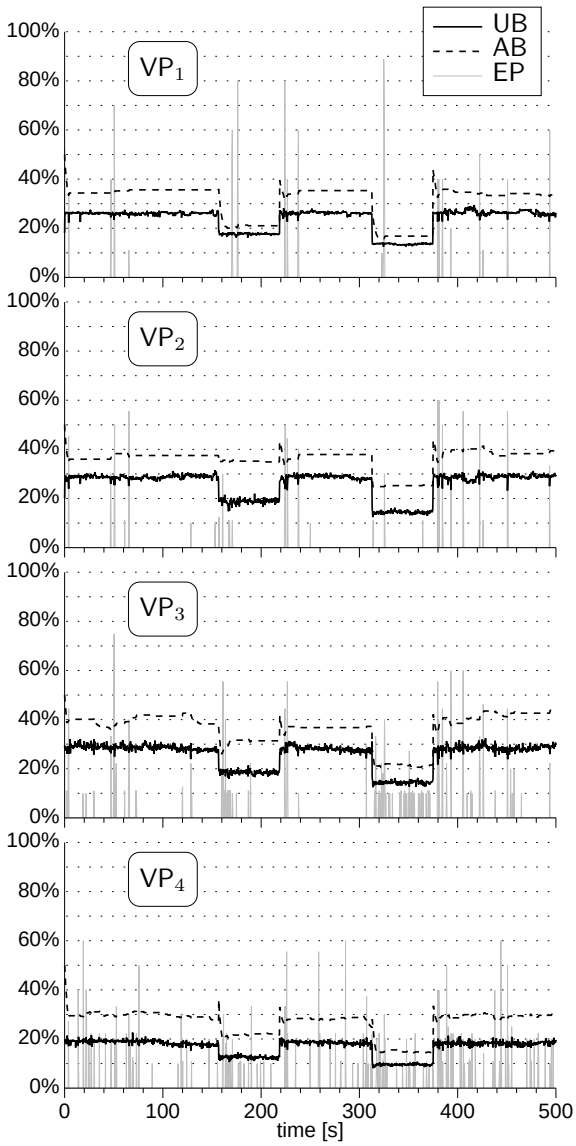

Figure 5: Bandwidth variation on the multi-core platform for the periodic pipeline.

guess of 50% for each of the four VPs is higher than the actual bandwidth consumption. Hence the bandwidth controller adapts the bandwidth accordingly (in this experiment we set $\mathsf{EP_s} = 0.1$, meaning that the RM allows up to 10% of budget exhaustion in each sampling interval).

At time 155, the arrival of another application forces the RM to lower the service level to 1. We can observe that the bandwidth is quickly adapted to the new status. At time 220, this last application completes allowing the pipeline application to go back to service level 0. A similar situation occurs in the interval $[315, 375]$. In this case, however, the incoming application demands a higher amount of computation that requires the RM to set the service level 2 for the periodic pipeline.

The second experiment is performed on an MPEG decoder, implemented as a distributed application consisting of an adaptive video server [7] and a playout client running on different machines connected over a wireless network. The video client is a modified VLC player that provides 3 different service levels with an overall bandwidth requirement of 80%, 11% and 8%, respectively. Figure 6 (top) reports assigned and used bandwidth (AB and UB) of the client application, together with the exhaustion percentage (EP). Initially, the client is the only application in the system, thus it runs at the highest service level (0). Since the assigned resources exceed the used resources by far, the feedback mechanism dynamically adjusts the assigned bandwidth. After about 25 seconds, another application starts up. The RM informs the client to switch to a lower service level and instructs the OS to shrink the client's reservation. In this case, some frames are skipped, as shown in Figure 6 (bottom). After 50 seconds, a third application starts up, so the RM enforces even stricter constraints on the video client, pushing the client to the lowest quality service level (2). At time 90, the RM sets the service level back to 1 due to the completion of the last application. Finally, at time 115, the initial condition is restored.

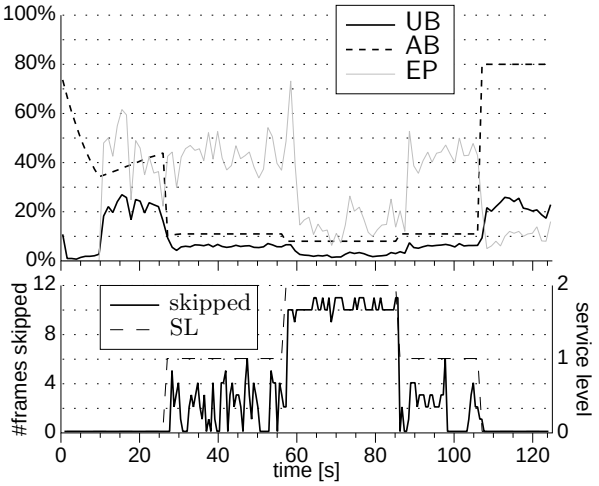

Figure 6: Bandwidth variation for the MPEG decoder.

6 Conclusions

This paper presented the approach developed in the ACTORS research project for managing computational resources over multi-core platforms. The adopted bandwidth reservation mechanisms allows the system designer to better control the allocation of the available resources, with respect to the classical threads and priorities approach. Together with appropriate measurements, this technique also facilitates automatic run-time resource management. The approach resulted to be very effective for handling time-sensitive applications with highly variable load.

The reservation mechanism is provided by a new Linux scheduling class, `SCHED_DEADLINE`, fully integrated in the 2.6.33 kernel release.

References

[1] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the* 19^{th} *IEEE Real-Time Systems Symposium*, pages 4–13, Madrid, Spain, Dec. 1998.

[2] L. Abeni, T. Cucinotta, G. Lipari, L. Marzario, and L. Palopoli. Qos management through adaptive reservations. *Real-Time Systems*, 29(2-3):131–155, 2005.

[3] E. Bini, G. C. Buttazzo, and M. Bertogna. The multy supply function abstraction for multiprocessors. In *Proceedings of the* 15^{th} *IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 294–302, Beijing, China, Aug. 2009.

[4] J. Bruno, J. Brustoloni, E. Gabber, B. Özden, and A. Silberschatz. Disk scheduling with quality of service guarantees. In *IEEE International Conference on Multimedia Computing and Systems*, volume 2, pages 400–405, Firenze, Italy, July 1999.

[5] D. Faggioli, F. Checconi, M. Trimarchi, and C. Scordino. An EDF scheduling class for the Linux kernel. In *Proceedings of the* 11^{th} *Real-Time Linux Workshop*, Dresden, Germany, Sept. 2009.

[6] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM Journal on Applied Mathematics*, 17(2):416–429, 1969.

[7] A. Kotra and G. Fohler. Resource aware real-time stream adaptation for MPEG-2 transport streams in constrained bandwidth networks. In *Proceedings of IEEE International Conference on Multimedia and Expo*, pages 729–730, Singapore, July 2010.

[8] G. Lipari and E. Bini. A methodology for designing hierarchical scheduling systems. *Journal Embedded Computing*, 1(2):257–269, 2005.

[9] N. Manica, L. Abeni, L. Palopoli, D. Faggioli, and C. Scordino. Schedulable device drivers: Implementation and experimental results. In *Proceedings of International Workshop on Operating Systems Platforms for Embedded Real-Time Applications*, Brussels, Belgium, July 2010.

[10] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves: Operating system support for multimedia applications. In *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, pages 90–99, Boston, USA, May 1994.

[11] A. K. Mok, X. Feng, and D. Chen. Resource partition for real-time systems. In *Proceedings of the* 7^{th} *IEEE Real-Time Technology and Applications Symposium*, pages 75–84, Taipei, Taiwan, May 2001.

[12] K. J. Nesbit, M. Moreto, F. J. Cazorla, A. Ramirez, M. Valero, and J. E. Smith. Multicore resource management. *IEEE Micro*, 28(3):6–16, May 2008.

[13] L. Rizvanovic and G. Fohler. The MATRIX: - a framework for real-time resource management for video streaming in networks of heterogenous devices. In *The International Conference on Consumer Electronics 2007*, Las Vegas, USA, Jan. 2007.

[14] D. Stiliadis and A. Varma. Latency-rate servers: A general model for analysis of traffic scheduling algorithms. *IEE/ACM Transactions on Networking*, 6(5):611–624, Oct. 1998.