Resource Management on Multicore Systems: The ACTORS Approach

Enrico Bini
Giorgio Buttazzo
Scuola Superiore Sant'Anna

Johan Eker
Ericsson Research

Stefan Schorr
Raphael Guerra
Gerhard Fohler
Technische Universität Kaiserslautern

Karl-Erik Årzén
Vanessa Romero Segovia
Lund University

Claudio Scordino
Evidence Srl

High-performance embedded systems require the execution of many applications on multicore platforms and are subject to stringent restrictions and constraints. The ACTORS project approach provides temporal isolation through resource reservation over a multicore platform, adapting the available resources on the basis of the overall quality requirements. The architecture is fully operational on both ARM MPCore and x86 multicore platforms.

Advanced cell phones show a strong Unix heritage—for example, support for processes and threads and separate memory spaces—but none of these features are exposed to the user. Most features are hidden even from application programmers, especially in some operating system mechanisms such as task scheduling. One main reason why multitasking in cell phone operating systems is restricted for third-party developers is resource management, which decides which tasks and applications get to consume CPU cycles, battery lifetime, and communication bandwidth. Cell phone manufacturers carefully tune the resource distribution and scheduling to optimize the user experience and then lock the APIs. Multitasking is typically limited to a number of APIs that provide services (such as audio streaming) running in the background. Most standard operating systems only provide crude mechanisms to distribute resources among applications, mainly using priorities. The priority assignment, however, prevents isolation in multiapplication systems, because it requires a system-wide knowledge of all applications and services competing for the same resources.

Resource reservations provide a more suitable interface for allocating resources such as CPU to a number of applications.[1,2] According to this method, a resource manager can assign a fraction of the platform capacity to each application, which runs as if it were executing alone on a less-performing virtual platform, independently of the other applications' behavior. In this sense, each application's temporal behavior isn't affected by the others and can be analyzed in isolation.

The concept of a virtual platform isn't new. Nesbit et al. introduced the virtual private machine, which provides an abstract view of all the physical resources (processors, memory bandwidth, and caches) along both

Published by the IEEE Computer Society

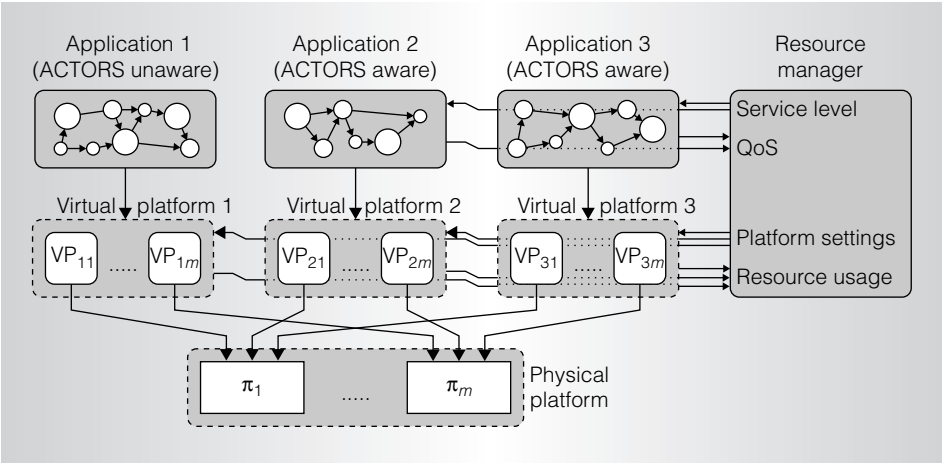

Figure 1. The ACTORS architecture. Applications are first allocated by the resource manager to individual virtual platforms (VPs), which are then mapped to physical cores. The resource manager adjusts both application and platform parameters to achieve the target QoS with minimal resource consumption.

spatial and temporal dimensions.[3] The abstraction for virtual platforms proposed in the ACTORS (Adaptivity and Control of Resources in Embedded Systems) project (http://www.actors-project.eu) instead provides a finer control of the CPU time that allows guaranteeing applications with real-time requirements. Indeed, abstracting only the CPU time introduces a degree of uncertainty because other kinds of resources weren't modeled. In the ACTORS project, we manage this uncertainty with a feedback loop that adjusts the CPU time according to the final quality of service (QoS) delivered to the user.

The ideas presented in this article were driven by the desire to automatically allocate the available resources, not only at design time, but also at runtime, based on actual user demand. For operating mechanical and electrical systems in uncertain environments, feedback control is a key technique, which in ACTORS is also adopted to control the 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 generally more complex and application dependent. In the ACTORS project, we focus on streaming applications—such as audio and video codecs or radio receivers and transmitters—that have an inherent notion of progress, because the workload is divided into a number of frames and there exists a nominal expected rate (for example, frames per second). Although the framework we present here isn't limited to streaming applications, it's particularly suited for the large class of devices performing real-time tasks, such as video decoding and streaming.

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

ACTORS software architecture

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

To perform resource allocation independently of a particular physical platform, ACTORS allocates applications to virtual platforms. A virtual platform consists of a set of *virtual processors*, each executing a portion of an application. ACTORS implements each virtual processor as a Linux reservation server allocated onto a physical core.

To cope with variability, or when the computational requirements aren't precisely

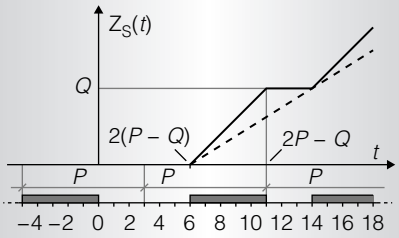

Figure 2. Supply function of a periodic server. Server's worst-case execution is represented by the gray schedule on the bottom timeline. The dashed line represents a linear lower bound of the supply function that enables a simpler analysis.

specified, applications notify the achieved QoS to a resource manager. The resource manager adapts the reservations on the virtual processors using a feedback loop that measures the resource actually consumed and notifies the applications about the newly assigned service levels. The system also allows executing applications that can't specify their QoS levels (ACTORS-unaware applications).

Resource abstraction

A key design requirement for simplifying portability between platforms is application development independent of physical platforms. This is crucial for multicore systems, where adding a processor might not necessarily improve performance.[4] In fact, embedded software that's developed to be highly efficient on a given multicore platform could be inefficient on a new platform with a different number of cores.

Roughly speaking, a multicore platform can be properly abstracted by two key features: the overall computing power α of the entire multiprocessor platform, and the number m of virtual processors of the platform. Although a fully parallelizable application is only affected by α (because it can exploit any degree of parallelism), a fully serial application can only execute on one core—so, for the same overall capacity α, the smaller m, the better.

ACTORS represents a virtual multicore platform as a set of m sequential virtual processors. We briefly recall the basic concepts used for abstracting a single processor.

Virtual processor abstraction

Abstracting a virtual processor requires a measure of the amount of computation the virtual processor can provide. For this purpose, the concept of *supply function* has been proposed.[5,6]

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

A supply function is nonnegative, 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 like the one in Figure 2. Unfortunately, abstracting a virtual processor by the pair (Q, P) is inappropriate if the actual implementation mechanism isn't a periodic server, but a static partition of time. Moreover, early in the design process, an abstraction should not rely on a specific implementation mechanism, such as the pair (Q, P).

In ACTORS, a virtual processor is modeled by using a *bounded delay abstraction*, introduced by Mok et al.[5], to represent a virtual platform on a wide class of possible implementations. A bounded delay abstraction can be fully described by two parameters: the bandwidth α, which measures the relative speed with which a resource is assigned to the demanding application, and a delay Δ, which represents the worst-case service delay. This abstraction is common to other fields, such as networking[7] and disk scheduling.[8] Thus, the analysis proposed here can also be extended to a more complex system including different types of resources.

Informally speaking, α and Δ describe the tightest linear lower bound to the supply function. In Figure 2, a dashed line represents such a linear lower bound. Given the supply function $Z_k(t)$ of the VP_k, we formally define the two parameters as

$$\alpha_k = \lim_{t \to \infty} \frac{Z_k(t)}{t} \quad (1)$$

$$\Delta_k = \sup_{t \geq 0} \left\{ t - \frac{Z_k(t)}{\alpha_k} \right\} \quad (2).$$

Indeed, the bandwidth captures a virtual processor's most significant feature. However, two virtual processors with the same bandwidth can allocate time differently: suppose that one virtual processor allocates the processor for 1 ms every 10 ms and another allocates the processor for 1 s every 10 s. Both virtual processors have the same bandwidth (10 percent of the physical processor), but the first virtual processor 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.[5] For example, a periodic server with period P and budget Q has the following α, Δ parameters:

$$\alpha = \frac{Q}{P} \quad \Delta = 2(P - Q) \tag{3}$$

as Figure 2 also illustrates.

Virtual multiprocessor abstraction

The abstraction of a virtual multiprocessor must also represent the degree of parallelism. Thus, a virtual multiprocessor is represented by a set of virtual processors,[9] 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 can guaranteed on top of this abstraction.[9] A parallel application is then partitioned into subsets, each mapped onto a single sequential virtual processor.

Linux implementation

Linux was initially designed as a general-purpose operating system for servers and desktop environments, so not much attention has been dedicated to real-time issues. In particular, the current scheduling framework—recently introduced by Molnar as a substitute for the previous O(1) scheduler[10]—does not contain any resource reservation mechanism capable of guaranteeing real-time constraints. Here, we explain 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 runtime, the scheduler core inspects the running tasks' policy and schedules each task by 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):

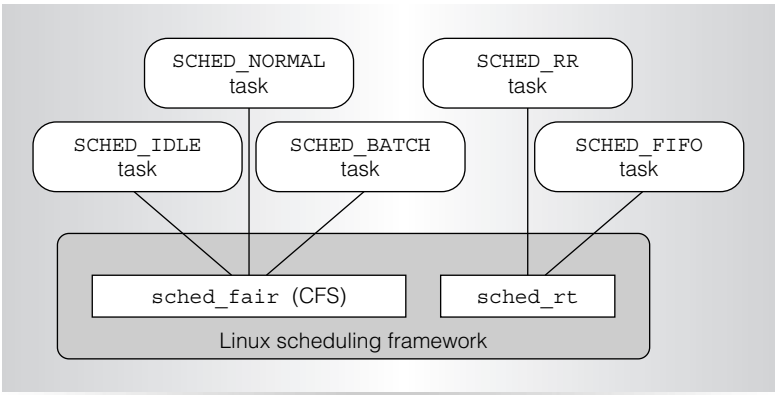

Figure 3. The Linux scheduler is a modular framework where new scheduling classes can be added. CFS (Completely Fair Scheduler) is one of these scheduling classes.

- `sched_fair` implements the Completely Fair Scheduler (CFS) algorithm and schedules tasks with `SCHED_OTHER` or `SCHED_BATCH` policies. Tasks 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 with `SCHED_FIFO` or `SCHED_RR` policies.

Unfortunately, none of these scheduling classes are suitable for reserving a share of the processor to tasks with timing constraints.[11] The API available on Linux, in fact, allows us to assign a share of processor time to a task but it is not designed to specify temporal constraints (such as deadlines or periods) to this share.

Therefore, the existing scheduling algorithms can't 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 can't be known in advance because it depends on the number of tasks running in the system at that time.

Usually, this problem is solved by overallocating the amount of CPU to time-sensitive tasks so that they receive the CPU more often, meeting their timing constraints. However, such a strategy is counterproductive, especially in the field of embedded systems, where reducing the amount of resources is often critical. Moreover, this strategy can't guarantee that tasks' timing constraints will be met in all circumstances.

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. The code is freely available on the public Git repository at http://gitorious.org/sched_deadline. The class adds the possibility of scheduling tasks using the Constant Bandwidth Server (CBS) without changing the behavior of tasks scheduled using the existing policies.[2] The CBS, in turn, is based on Earliest Deadline First (EDF) task queues. The implementation doesn't make restrictive assumptions on the tasks' characteristics.

In recent years, OS developers have proposed several real-time schedulers for Linux, but none of these became part of the official Linux kernel. Our approach differs from existing schedulers in several aspects:

- It's integrated with the mainline Linux scheduler.
- It's a new, self-contained scheduling class instead of a kernel module invoked through hooks in the main scheduling class.
- It natively supports multicore platforms.
- It's not architecture dependent, so it can run on all platforms Linux already supports.

Extensive experiments showed that tasks executed under `SCHED_DEADLINE` instead of `SCHED_FIFO` can better meet their timing requirements.[12] For example, when `SCHED_FIFO` is used to schedule two instances of a GTK video player together with other CPU-intensive tasks, only one instance of the player can meet its timing constraints (that is, 25 frames per second). When `SCHED_DEADLINE` is used instead, both instances can reproduce the movie smoothly, and each instance's performance is proportional to the share of CPU assigned through the real-time parameters.

User-level interface

In ACTORS, the `SCHED_DEADLINE` class has been slightly modified to meet the project's specific needs, such as adding hierarchical scheduling handled through the standard Linux mechanisms (that is, control groups). The modified scheduling class has been internally called `SCHED_EDF`.

A new system call, `sched_setscheduler2`, allows application programmers to create or modify the task's budget and period. The system administrator must provide these values also for aperiodic tasks, so that they are constrained to execute no longer than their budget within each period, thus ensuring a proper isolation between the running tasks.

For periodic tasks, 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 current period ends. The task itself must invoke this system call at the end of each period. If the task belongs to a virtual processor, the other tasks in the same virtual processor can continue their execution.

Virtual processors can be created and destroyed using the same control groups (*cgroups*) interface available for the other scheduling classes—that is, through the *mount* Linux command. The cgroups interface also allows programmers to bind tasks to a virtual processor and set or change its reservation parameters. A virtual processor's budget and period can be set using two entries in the cgroups file system (`cpu.edf_runtime_us` and `cpu.edf_period_us`, respectively). These files are created once the cgroup file system is mounted.

The system maintains a hierarchy of virtual processors. For this reason, without

setting the budget and period of the "root" virtual processor, the other virtual processors can't 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 runtime behavior of the group of tasks. In particular, it contains the amount of budget the virtual processor has used so far; the number of times the virtual processors' budget has been recharged; and the number of times the virtual processor entered the "hard reservation" state (that is, the virtual processor has been blocked because it finished the whole budget available in a period).

Our deadline scheduler's user-level interface is straightforward and can be used by a user-level component to configure the system at runtime, according to the actual CPU request of each running application. In our project, this logic has been implemented inside a component called the Resource Manager.

Resource Manager

In ACTORS, the Resource Manager (which is implemented as a user-level application) is responsible for allocating the resources to applications. 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, where application demands are abstracted as a small set of service levels, each characterized by a QoS and resource requirements.[13] In this way, only significant changes trigger a system reconfiguration.

Service-level assignment

Table 1 shows an example of an application that can operate at four service levels using four virtual processors. In the table, SL, QoS, α, Δ, and BWD denote the service-level index, quality of service, total bandwidth, tolerable application delay, and bandwidth distribution over the individual virtual processors, respectively. α is expressed as a percentage of a single core's computational power. 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 runtime by a feedback loop.

Table 1. Service-level table of an application.

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]

*SL: service-level index; QoS: quality of service; α: total bandwidth; Δ: tolerable application delay; BWD: bandwidth distribution.

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

Once service levels are assigned, the resource manager distributes the bandwidth. Virtual processors are created through the Linux interface we described earlier. The server's budget and period are computed based on the corresponding (α, Δ) parameters, as described in Equation 3. The mapping of virtual processors is also formulated as an ILP problem.

Resource adaptation

Resource adaptation is achieved through a control mechanism, which uses a combination of feed-forward and feedback strategies to modify resource allocation at runtime.[14] The service-level assignment and bandwidth distribution constitute the feed-forward part of the resource allocation strategy. The feedback part consists of one bandwidth controller for each virtual processor. The bandwidth controller checks whether the virtual processor's tasks are making optimal use of the allocated bandwidth and acts to avoid wasting resources without degrading the application's performance. Bandwidth controllers are executed periodically with a period that's a multiple of the period of the virtual processor they're controlling.

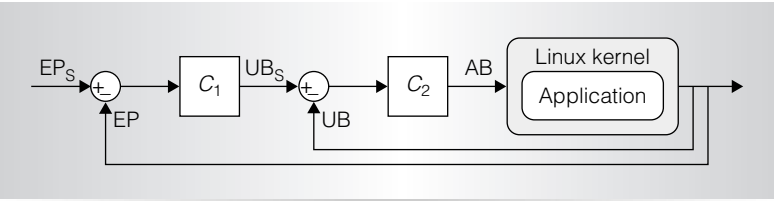

Figure 4. Bandwidth controller's structure. The controller C_1 defines the new values of the set point for the used bandwidths. The controller C_2 adjusts the virtual processor bandwidth by modifying the assigned bandwidth. (UB: used bandwidth; AB: assigned bandwidth; EP: exhaustion percentage.)

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

- *used bandwidth*, the average used budget over the controller's sampling period, and
- *exhaustion percentage*, the percentage of server periods over the last sampling interval in which the virtual processor budget is totally consumed. This indicates that the application was likely to require more bandwidth.

The bandwidth controllers have a cascade structure (see Figure 4). The controller affects the application behavior through the Linux interface. Exhaustion percentages denote the maximum percentage of budget exhaustion that the application can tolerate.

The feed-forward/feedback structure employed in ACTORS is based on the Aquosa architecture,[15] employed in the European Commission's Framework for Real-Time Embedded Systems based on Contracts (EC Frescor) project. The main extensions developed in ACTORS are the support for multicore applications and for hard rather than soft reservations.

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 virtual processor. Each of the four virtual processors (VP_1, VP_2, VP_3, and VP_4) is bound over a different physical processor. The first stage is activated periodically. The next stages are activated when the preceding ones are complete. The application has four service levels, reported in Table 1.

Figure 5 shows the evolution over time of the used bandwidth, the assigned bandwidth, and the exhaustion percentage. At time 0, the periodic pipeline application registers with the resource manager, which assigns the highest service level (0) because no other applications are running in the system. The initial bandwidth guess (50 percent for each of the four virtual processors) is higher than the actual bandwidth consumption. Hence, the bandwidth controller adapts the bandwidth accordingly (in this experiment, we set EPs = 0.1, meaning that the resource manager allows up to 10 percent of budget exhaustion in each sampling interval).

At time 155, another application arrives, forcing the resource manager to lower the service level to 1. We can observe that the bandwidth adapts quickly 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, which requires the resource manager to set the service level 2 for the periodic pipeline.

The second experiment is performed on a MPEG decoder; it's implemented as a distributed application consisting of an adaptive video server[16] and a playout client running on different machines connected over a wireless network. The video client is a modified VLC player that provides three service levels with an overall bandwidth requirement of 80 percent, 11 percent, and 8 percent, respectively. The top of Figure 6 reports the client application's assigned and used bandwidth, together with the exhaustion percentage. Initially, the client is the only application in the system, so it runs at the highest service level (0). Because the assigned resources exceed the used resources, the feedback mechanism dynamically adjusts the assigned bandwidth. After about 25 seconds, another application is activated. The resource manager informs the client to switch to a lower service level

and instructs the operating system to shrink the client's reservation. In this case, some frames are skipped, as shown at the bottom of Figure 6. After 50 seconds, a third application starts up, so the resource manager enforces even stricter constraints on the video client, pushing the client to the lowest quality service level (2). At time 90, the resource manager sets the service level back to 1 because the last application is complete. Finally, at time 115, the initial condition is restored.

The ACTORS project's main contribution was the development of a resource manager for the automatic partition of time-sensitive applications over multicore platforms. The software support has been provided on the Linux operating system, which has been extended with a resource reservation mechanism on top of a deadline-based scheduler.

As the trend toward heterogeneous multicore becomes more dominant for high-performance computing, a future challenge will be to extend the proposed framework over heterogeneous platforms that include specialized or configurable hardware, possibly geographically distributed on different servers. MICRO

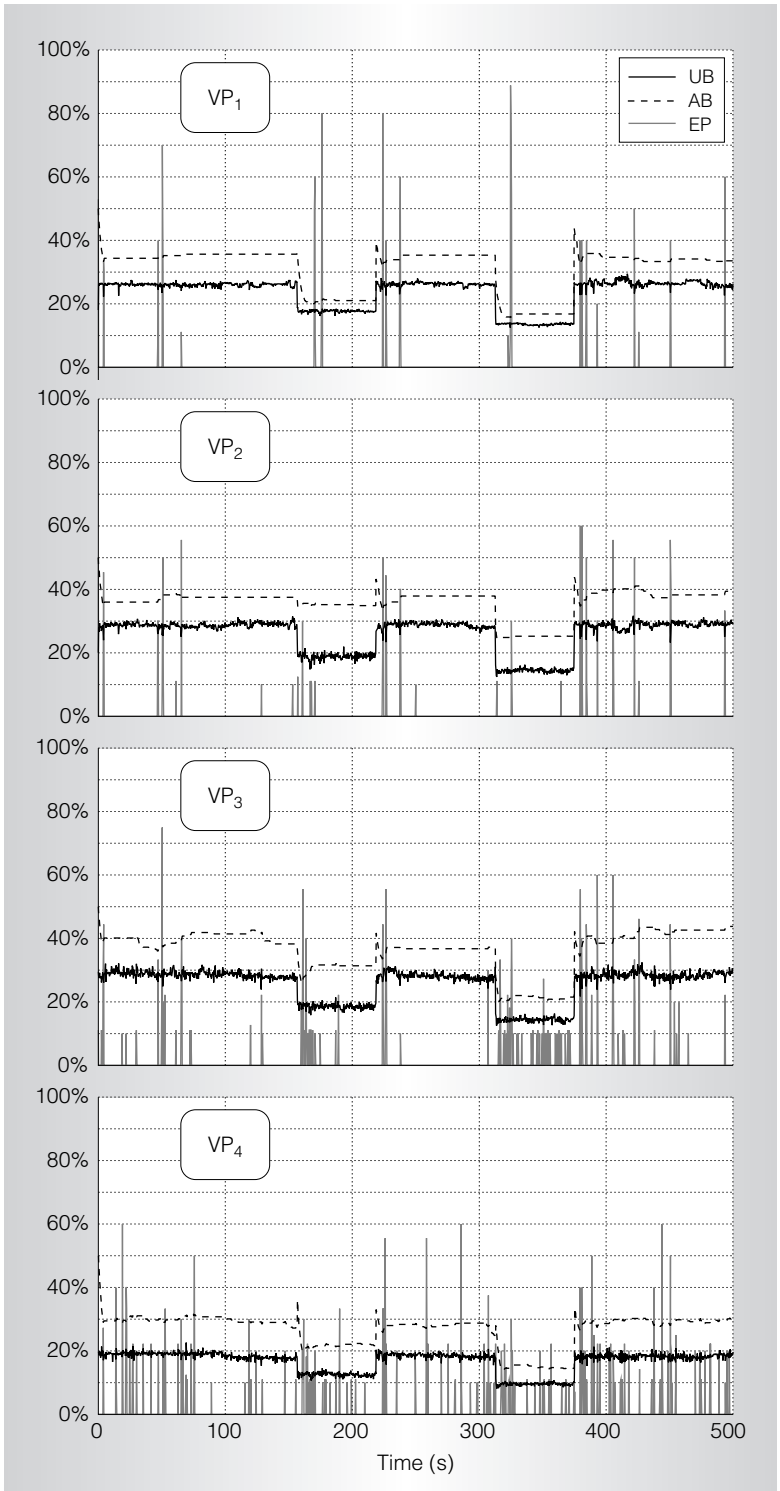

Figure 5. Bandwidth variation on the multicore platform for the periodic pipeline. The *x*-axis reports the simulation time in seconds.

Acknowledgments

The European Commission partially supported this work under the ACTORS project (FP7-ICT-216586).

References

1. C.W. Mercer, S. Savage, and H. Tokuda, ''Processor Capacity Reserves: Operating System Support for Multimedia Applications,'' *Proc. IEEE Int'l Conf. Multimedia Computing and Systems,* Carnegie Mellon Univ., 1994, pp. 90-99.
2. L. Abeni and G. Buttazzo, ''Integrating Multimedia Applications in Hard Real-Time Systems,'' *Proc. 19th IEEE Real-Time Systems Symp.,* IEEE CS Press, 1998, pp. 4-13.
3. K.J. Nesbit et al., ''Multicore Resource Management,'' *IEEE Micro,* vol. 28, no. 3, 2008, pp. 6-16.

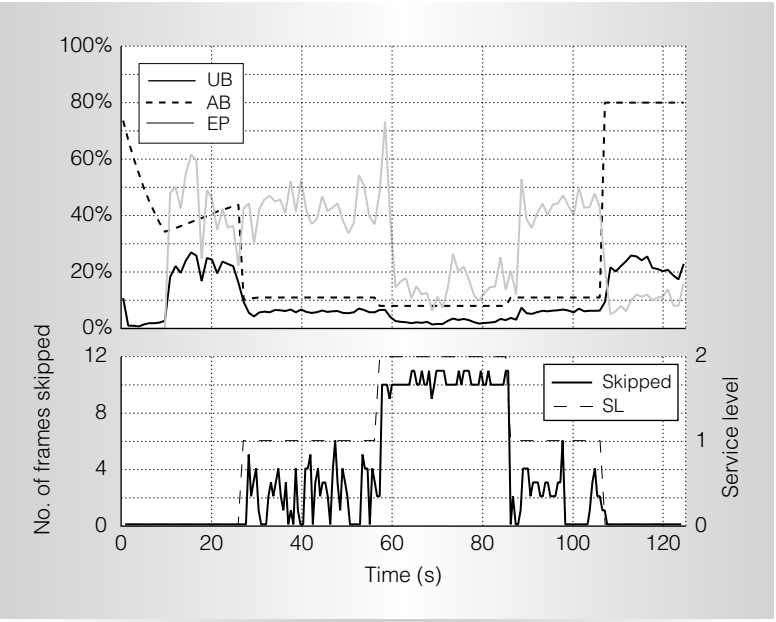

Figure 6. Bandwidth variation for the MPEG decoder. The figure gives an overview of the client's service level, used and assigned bandwidth, and exhaustion percentage. It also shows the number of skipped frames in the adapted stream. As the service level is reduced, the number of frames skipped increases.

4. R.L. Graham, ''Bounds on Multiprocessing Timing Anomalies,'' *SIAM J. Applied Mathematics,* vol. 17, no. 2, 1969, pp. 416-429.
5. A.K. Mok, X. Feng, and D. Chen, ''Resource Partition for Real-Time Systems,'' *Proc. 7th IEEE Real-Time Technology and Applications Symp.,* IEEE CS Press, 2001, pp. 75-84.
6. G. Lipari and E. Bini, ''A Methodology for Designing Hierarchical Scheduling Systems,'' *J. Embedded Computing,* vol. 1, no. 2, 2005, pp. 257-269.
7. D. Stiliadis and A. Varma, ''Latency-Rate Servers: A General Model for Analysis of Traffic Scheduling Algorithms,'' *IEEE/ACM Trans. Networking,* vol. 6, no. 5, 1998, pp. 611-624.
8. J. Bruno et al., ''Disk Scheduling with Quality of Service Guarantees,'' *IEEE Int'l Conf. Multimedia Computing and Systems,* vol. 2, IEEE Press, 1999, pp. 400-405.
9. E. Bini, G. Buttazzo, and M. Bertogna, ''The Multi Supply Function Abstraction for Multiprocessors,'' *Proc. 15th IEEE Int'l Conf. Embedded and Real-Time Computing Systems and Applications,* IEEE Press, 2009, pp. 294-302.
10. I. Molnar, ''Modular Scheduler Core and Completely Fair Scheduler (CFS),'' https://lkml.org/lkml/2007/4/13/180.
11. D. Faggioli et al., ''An EDF Scheduling Class for the Linux Kernel,'' *Proc. 11th Real-Time Linux Workshop,* 2009, pp. 197-204.
12. N. Manica et al., ''Schedulable Device Drivers: Implementation and Experimental Results,'' *Proc. Int'l Workshop on Operating Systems Platforms for Embedded Real-Time Applications,* Politécnico do Porto, 2010, pp. 53-62.
13. L. Rizvanovic and G. Fohler, ''The MATRIX: A Framework for Real-Time Resource Management for Video Streaming in Networks of Heterogenous Devices,'' *Int'l Conf. Consumer Electronics 2007,* IEEE Press, 2007, doi:10.1109/ICCE.2007.341500.
14. V.R. Segovia et al., ''Processor Thermal Control Using Adaptive Bandwidth Resource Management,'' to be published in *Proc. 18th World Congress Int'l Federation of Automatic Control,* Elsevier, 2011, pp. 123-129.
15. L. Abeni et al., ''QoS Management through Adaptive Reservations,'' *Real-Time Systems,* vol. 29, nos. 2-3, 2005, pp. 131-155.
16. A. Kotra and G. Fohler, ''Resource Aware Real-Time Stream Adaptation for MPEG-2 Transport Streams in Constrained Bandwidth Networks,'' *Proc. IEEE Int'l Conf. Multimedia and Expo,* IEEE CS Press, 2010, pp. 729-730.

Enrico Bini is an assistant professor in the Real-Time Systems Laboratory at the Scuola Superiore Sant'Anna of Pisa. His research interests include real-time scheduling, embedded systems design, and optimization methods. Bini has a PhD in computer engineering from Scuola Superiore Sant'Anna. He's a member of the IEEE Computer Society.

Giorgio Buttazzo is a full professor in the Department of Computer Engineering at the Scuola Superiore Sant'Anna of Pisa. His research interests include real-time systems, scheduling and resource management algorithms. Buttazzo has a PhD in computer engineering from the Scuola Superiore Sant'Anna. He has coauthored six books on real-time systems, including *Soft Real-Time*

Systems: Predictability vs. Efficiency (Springer, 2010). He's a senior member of IEEE.

Johan Eker is a senior specialist at Ericsson Research. He's also the coordinator of the European Seventh Framework Program (FP-7) research project ACTORS. His research interests include embedded software with a focus on real-time systems and program language design. Eker has a PhD in automatic control from Lund University. He's a member of ACM.

Stefan Schorr is a doctoral candidate in real-time systems at Technische Universität Kaiserslautern. His research interests include real-time scheduling theory and multicore real-time systems. Schorr has an MSc in information technology from Technische Universität Kaiserslautern.

Raphael Guerra is a doctoral candidate in real-time systems at Technische Universität Kaiserslautern. His research interests include adaptive and flexible real-time systems. Guerra has an MSc in parallel and distributed computing from the Fluminense Federal University in Brazil.

Gerhard Fohler is a professor in real-time systems at Technische Universität Kaiserslautern. His research interests include adaptive real-time systems and networks and media processing. He received his PhD in computer science from the Vienna University of Technology. He is an associate editor of the *Real-Time Systems Journal* and a real-time systems editor of the *Journal of System Architectures*. He's chairman of the Euromicro Technical Committee on Real-Time Systems and a member of the IEEE Technical Committee on Real-Time Systems. He's a senior member of IEEE.

Karl-Erik Årzén is a professor in automatic control at Lund University. His research interests include embedded control, control of computing systems, real-time systems, cosimulation tools, and adaptive resource management. Årzén has a PhD in automatic control from Lund University. He's a member of IEEE.

Vanessa Romero Segovia is a doctoral student in automatic control at Lund University. Her research interests include resource management in real-time systems and control process. Romero got her MS in electrical engineering in the field of automation and control from Technische Universität Kaiserslautern. She's a member of IEEE.

Claudio Scordino is a software engineer and project manager at Evidence Srl. His current research activities include energy-aware scheduling, real-time operating systems, and embedded devices. Scordino has a PhD in computer science from the University of Pisa.

Direct questions and comments about this article to Enrico Bini, Scuola Superiore Sant'Anna, Via G. Moruzzi 1, 56127 Pisa, Italy; e.bini@sssup.it.