Chapter 6
Run-Time Resource Management at the Operating System level

Patrick Bellasi, Simone Corbetta, William Fornaciari

Abstract Available hardware platforms provide the applications with an extended set of physical resources, as well as a well defined set of power and performance optimization mechanisms (i.e., hardware control knobs). The software stack, meanwhile, is responsible of taking direct advantage of these resources, in order to meet application functional and non-functional requirements. The support from the Operating System (OS) is of utmost importance, since it gives opportunity to optimize the system as a whole.
Purpose of this chapter is to introduce the reader to the challenge of managing physical and logical resources in a complex multi- and many-core architectures, focussing on emerging MPSoC platforms.

6.1 Introduction

Modern applications are of a wide variety, and they all have different requirements in terms of hardware and software resources, performance goals, power and energy constraints. What we are experiencing in today's electronics is a continuous convergence of different application classes into the same environment: life-critical, real-time, mobile and general processor power are converging to the same System-on-Chip (refer to Figure 6.1).

The increasing demand for applications and scenarios translates into an increasing demand for processing power and integration, as it can be seen from Figure 6.2. This adds several challenges to the design of such applications. In addition, to cope with design and implementation costs, a suitable design methodology should contemplate the re-use of part of the subsystems. In this context, the Resource Man-

Contact Author: Patrick Bellasi
Dipartimento di Elettronica e Informazione, Politecnico di Milano, Milano, Italy
e-mail: `bellasi@elet.polimi.it`

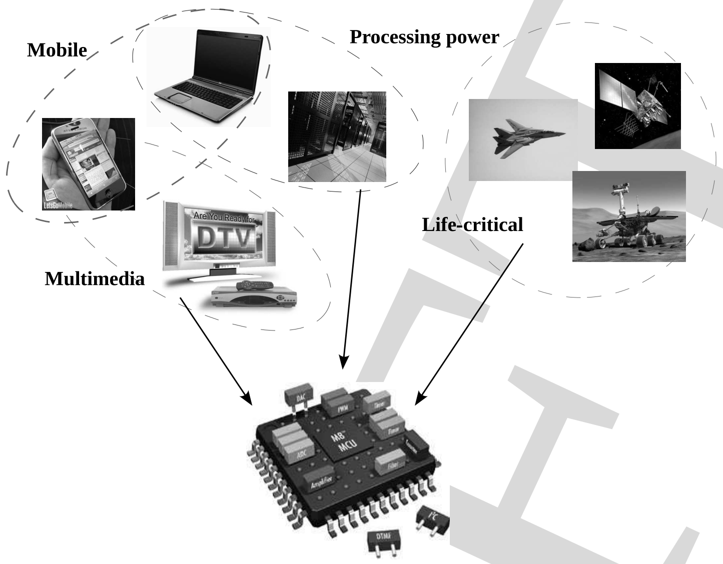


Fig. 6.1 Different applications scenarios are converging to the same SoC design paradigm.

ager (RM) should be able to operate on slightly different applications, with different resources and with as little as possible porting efforts.

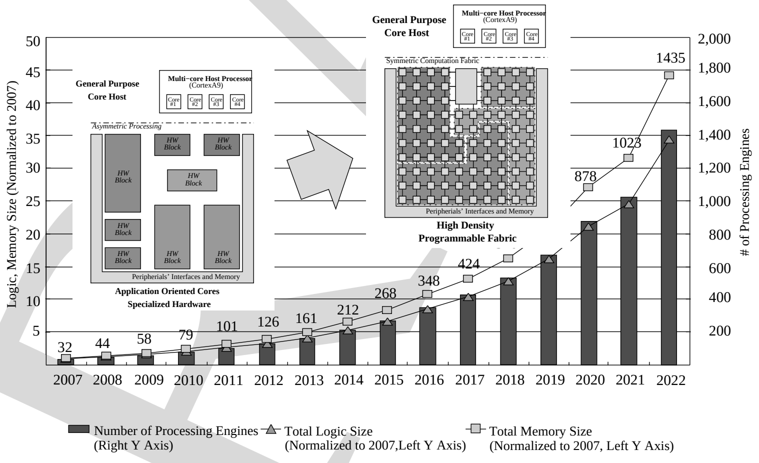


Fig. 6.2 Increasing computing power density brings to ever more complex architectures with an increasing count of per-chip functionality and resulting in more complex applications. The chart represents the PE density per technology year, taken from [1].

Without any loss in generality, we can point out two main broad classes: *high performance* and *low power*. High performance encloses all applications for which high operating frequency and high throughput are experienced, while low-power applications are those for which energy budgeting is of major importance. Notice that speaking of "power" or "energy" is not the same, and the use of one or other term relies on the target application scenario. Refer to Section 6.4 for a more detailed discussion on this distinction. In conjunction, technology and design advances lead to a proliferation of mobile embedded systems, for which several additional challenges raise up. The combination of the aforementioned requirements and the application platforms (i.e., mobile against non-mobile), gives us free room for a simple yet comprehensive classification of the interested scenarios in the MPSoC domain. This fact is graphically shown in Figure 6.3.

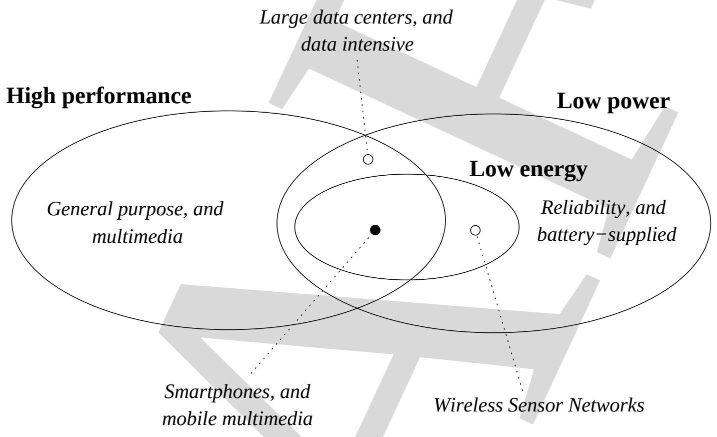

Large data centers, and data intensive
High performance
Low power
Low energy
General purpose, and multimedia
Reliability, and battery−supplied
Smartphones, and mobile multimedia
Wireless Sensor Networks

Fig. 6.3 Application domains are mapped to their respective design requirements. According to the required power/performance ratio, we can detect several application domains, in both the mobile and non-mobile case.

Five slightly different domains have been detected. General high-performance applications comprise both mobile and non-mobile domains. In the non-mobile case, for instance, we can find desktop personal computers, where the user experience from the performance view-point is of utmost importance; along with those requirements, such applications are mainly general purpose, meaning that they are designed to perform a wide spectrum of jobs, without precise optimization of power or energy aspects. Nevertheless, it is true that there exist cases in which high performance and low power are a common goal, as in systems collecting sensible data, such as large data centers; in this case the peak power has to be minimized, in order to reduce cooling costs, increase system reliability, and decrease device failure rate. General low-power operations are required by almost all mobile embedded systems, but different approaches should be taken while considering high performance and energy savings domains. Mobile multimedia devices (e.g. Smartphones) fall in this category. The effective trade-off between very high performance, for instance to ensure audio and video capabilities, and ultra low-power operation is a challenging task. Last, applications for which energy savings are of primarily relevance but in which performances can be lower than in any other case exist: wireless nodes in a Wireless Sensor Network belong to this set.

The remainder of the chapter is organized as follows: Section 6.2 gives an insight of the problem of managing resources at run-time, while discussion on the support from the OS is given in Section 6.3. A more detailed review of the existing Power Management (PM) frameworks based on the Linux kernel are presented later on in Section 6.4. Conclusions are given in Section 6.6.

6.2 Run-Time Resource Management (RTRM)

So far we have seen how modern MPSoC architectures provide an exceptional flexibility, with a huge set of physical and logical resources. The presence of multiple applications that can be potentially integrated within the same chip makes the problem of managing such resources a challenging problem. By “Run-Time Resource Management (briefly, RTRM)” we hereby mean the set of processes, techniques, methodologies and instruments that allow to *use* the available resources, provided an objective function. Generally speaking, resource utilization is subject to specific objectives and constraints, depending on both the effective hardware implementation and the considered application. One of the objective of the Resource Manager is thus to ensure that all such constraints and requirements are met, while satisfying the Quality-of-Service (QoS). A RTRM will perform this action dynamically, because there will be no chance to know which will be the applications (and, as a consequence, the requested resources) a priori.

Purpose of this section is to introduce in a more formal way the concepts related to the problem of managing the resources, the run-time component and the role of the Runtime Manager.

6.2.1 Problem overview

The concept of *resource* is at the same time enough general to hide some details on its real nature (e.g., physical or logical implementation) and enough detailed to allow us to employ it in a constrained optimization problem. Thus, a generic resource can either refer to a processor, a memory subsystem in a memory hierarchy or it can also refer to the computation time fraction assigned to a task. Either ways, each resource has a precise meaning within its own context. The set of available resources in a system can be defined as the set $R = \{r_1, r_2, ..., r_N\}$ of all those N resources we have access to. Each resource r_j in R can be of any kind, and we can further assume that there exist several *types*, that are clearly dependent on the context. $C = \{c_1, c_2, ..., c_M\}$ is the set of M types (or classes) of resources. A static mapping from the resources to the corresponding classes exist, and it can be meant as being (quasi) independent on the target application. Such mapping $M \subseteq R \times C$ is defined by an ad-hoc function $map(r_i, c_j)$, taking as inputs a generic resource $r_i \in R$ and a generic class $c_j \in C$. At any instant of time, only a subset $AR \subset R$ of the resources is available, and represents the resources that are not in use by any other task in the system. Complimentary, the set $BR = AR^c$ keeps track of the used (busy) resources. Considering the set $T = \{t_1, t_2, ..., t_P\}$ of P tasks in the system, there will exist a mapping from the system resources to the tasks: $U = \left\{u \in R \times T \;\; and \;\; u = uses(r_i, c_j), \;\; with \;\; r_i \in R, c_j \in C\right\}$. The obvious relation $U \perp R = BR$ holds, where the $\perp$ operator is the projection operator, returning the set of resources from a collection of pair-like sets.

The choice U of how to map resources to tasks, i.e. the implementation of the *uses* function depends upon a specific objective function. Such objective function gives priorities to specific resources for requesting tasks, and such priority changes according to the objective function. Generally, the objective function can be modeled through a set of metrics: performance, memory throughput, execution latency, power consumption, and so on. Thus, roughly speaking, the role of the Resource Manager is to fill in the set U with the appropriate entries, according to the *current* implementation of the *uses* function. Such implementation can change at run-time, for instance due to changing workloads or application requirements. Hence, there is a need for a Run-Time Resource Manager component, that can cope with the changing application scenario, resources availability and requirements.

6.2.2 Resource manager overview

Not every RTRM is suitable for each application/platform configuration. Several differences are experienced passing from an application to the other, and so the mapping would change. For this reason, a minimal set of requirements have to be guaranteed from the RTRM view-point. The following presents a subset of such requirements;

1. **Flexibility**, an RTRM should be able to work with the expected specifications under different scenarios. Flexibility ensures the reuse of the overall methodology, impacting positively on the design and implementation costs of the entire project. In addition, flexibility leads multiple applications to efficiently use the resources, without incurring in unwanted overheads;
2. **Scalability**, RTRM should work well for an increasing number of cores and, in general, of active resources. This is dictated by the growing and broad convergence toward multi- and many-core architectures. Scalability is hereby intended from two view-points: performance (latency, overhead) and energy efficiency (it should not consume more power than required, otherwise benefits will be outperformed);
3. **System-wideness**, decisions made by the manager should not interfere with other decisions previously taken, i.e. assigning specific resources to a newly incoming

task should not impact negatively on the already running tasks. In order to ensure this requirement, the RM should have system-wide perspective of what's going on in the system at any instant of time.

The previous list shows which are the main requirements for a Resource Manager to be suitable for the job. The effective implementation of the RM depends on several requirements related to the hardware and software architectures. Before giving an overview of the main components defining the manager, we define which are the classes of resource managers that can be found. This is just an adoption of the work done in [16]. The various classes of resource management can be identified according to the following metrics: static versus dynamic, hardware versus software implementation, centralized versus distributed, adaptive versus static.

Static resource managers are somehow hard-coded in the system software, in the sense that the decisions taken are defined in advance, according to a predefined policy. Policies cannot be changed at run-time so no flexibility is achieved in this case. In contrast, *dynamic* approaches let the managing process evolve according to the needs. The policies as well as other useful parameters might be changed during execution time, either explicitly (i.e., user-driven) or implicitly (i.e., guided by the current scenario).

The implementation can be either hardware, software or a smart mix of the two. Purely *software* approaches are very flexible, but they incur in high overhead; purely *hardware* approaches, on the other hand, provide high performance at the cost of flexibility. A mixed *hardware/software* co-design can optimize the performance and the flexibility. The real challenge is in finding which part will be in hardware and which part will be in software. Examples of purely software or hardware approaches are reported in [16].

Centralized approaches provide a single entity that gathers all the necessary information from the underlying hardware and from the upper software. *Distributed* approaches, instead, provide multiple control points, that gather local information. The former technique lacks of scalability as well as efficiency, since with a huge amount of resources to handle, the number of information to be gathered would increase exponentially. On the other hand, distributed approaches have only local view of the problem, and thus do not provide a system-wide optimization point-of-view. For these reasons, a more suitable approach should have a *hierarchical* control, i.e. integrating both local- and system-wide views of the same problem.

Last, decisions should be taken with respect to an adaptive or non-adaptive mechanism. *Adaptive* mechanism can follow the workload changes in the system, and they are clearly dynamic in nature. In contrast, *non-adaptive* (also called *fixed*) approaches are of a static nature and do not adapt their behavior to the changing context.

The selection of the implementation details of the aforementioned design parameters affects how the manager will behave in the target system. The most interesting approaches reported in [16] can thus be classified as shown in Figure 6.4.

OMAP platform from Texas Instruments [5] provides a software and centralized/distributed implementation, in which a single master Resource Manager resides

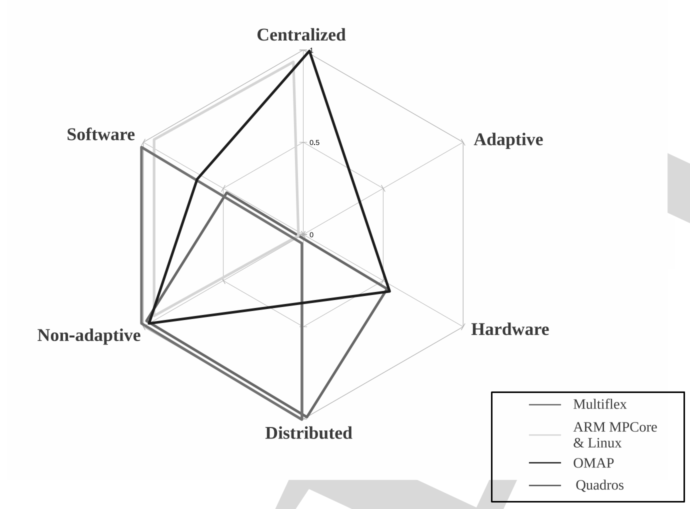


Fig. 6.4 Different commercial approaches are hereby reported comparing different features for an RTRM. Data is taken from [16]. For each interesting feature, a value of 0 stands for no support, a value of 1 stands for full support, while a value in between (e.g., 0.5) means the feature is only partially supported. This is especially useful while comparing hardware and software implementations.

on the central general-purpose processor (the "host"), and a slave RM server resides on each system co-processor, e.g. DSP. The role of the master RM is to manage and allocate slave DSPs, create tasks and allocate communication resources.

The approach followed by RTXC RTOS is based on the replication of RTOS kernel services on each core of the target platform containing MPSoC and multiple DSP coprocessors. However, there is no actual RM as we might think, since it is in charge of the designer to decide for the allocation of tasks. This happens through the use of the RTLib, but no run-time adaptation is provided [16].

The ARM MPCore is a multiprocessor platform for general-purpose operating systems like Linux. In this context, Linux would act in SMP fashion, hiding the fact that multiple Processing Elements (PEs) are present. In this way, the RM is embedded in the underlying OS, and transparently provides to the applications a multi-core environment with several available resources to speed-up application execution.

Last, Multiflex from STMicroelectronics addresses Nomadik platforms [21] containing multiple general purpose processors executing either Linux or Symbian in SMP configuration, and several customized application processors (ASIP) for video, audio and 3D processing. Resource management is performed through a master-slave configuration, like the one presented for OMAP platforms. Coprocessors are effectively seen as *devices* where to push tasks for execution.

6.2.3 Run-time manager components

This section presents a bird's-eye-view on the main components that a Resource Manager is made by. This section provides a well-defined set of the components *specifying* an RM, yet from a high-level perspective. This has been inspired by the work in [16], presenting a valid, but yet incomplete, modeling of the RM components and subsystems.

The core of an RTRM consists of the subsystem taking decisions about the resource-to-task allocation problem; in other words, the core of an RTRM consists of the implementation of the *uses* function. Such implementation resides in a separate component, that takes as input (also) the asserted constraints on the platform. Those constraints come from the software applications, and represent the "contract" between the software and the hardware. The effective allocation of the resources happens according also to the (current) objective function in the system. The objective function, as we said in Section 6.2.1, defines which are the candidate resources for the requesting application. A *QoS Estimator* component is employed for the sole purpose of estimating the potential resource allocation, so that to compare it with the application requirements. If an appropriate resource-to-task mapping is found, then the allocation is given back to the application, that can now begin the desired execution. The concept of "resource configuration" conveys in a single entity any kind of resources: physical resources (e.g., assigned cores), logical resources (e.g., clock cycles) and non-functional aspects (e.g., voltage/frequency selection for power-constrained systems). For this reason, we may want to call such configuration an *Operating Point*. In this perspective, the selection is performed by an *Operating Point Selection* module. Such module implements the core of the RM. It is based on two entities: *mechanisms* and *policies*. Mechanisms represent the observation and control knobs that the underlying hardware (or the HAL software) provides for the desired purpose, while policies define *how* resources to be allocated are selected.

From the description above, it is clear how an RTRM stays in the middle: it takes software-related and application-specific constraints and retrieves information from the underlying hardware platform. An overview of the RTRM in a system-wide perspective is shown in Figure 6.5. Three different aspects are then taken into account: *application*, *non-functional* and *platform* aspects. Application aspects refer to application requirements and constraints. Non-functional aspects collect those QoS-related, such as power consumption, throughput, memory bandwidth; these can be used directly to setup the objective functions. Last, platform come from the hardware: available resources, hardware architecture features, and so on.

6.3 Operating System support

In single-user single-tasking systems, when all the available resources were under direct control of the user, resource control was straightforward. Instead, with the advent of multi-tasking systems, the need for more complex resource control mech-

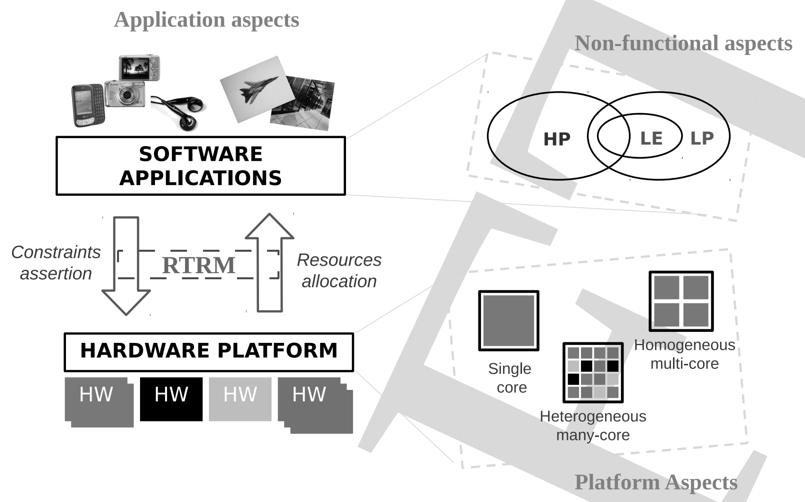


Fig. 6.5 The RTRM takes into consideration application aspects, non-functional aspects and platform aspects.

anisms raised. In these systems, one of the main goals is to control resources by sharing them fairly among all concurrent tasks. This requires the implementation of suitable "accounting mechanisms" and a "scheduling algorithm". Accounting mechanisms are used to keep track of the available resources and their allocation to requesting clients. Instead, scheduling algorithms should support the contrasting requirements of fairness and granted access to resources to all the clients.

This problem becomes particularly complex if we consider systems with a mix of best-effort and critical workloads. Regardless of their nature, critical workloads are characterized by the requirement of a certain amount of resources to grant their functionalities. Conversely, best-effort workloads could tread performances for resources without compromising functionality. In this scenario, a properly designed run-time resource manager should be aware of these differences and provide an adequate support for priority access to available resource.

To tackle the resource management problem, different architectural alternatives have been evaluated. Both middleware and native operating system support have been developed and quantitatively evaluated, especially on the specific context of multimedia applications [23]. In the rest of this section we focus on the native operating system approach, highlighting its role on the resolution of the resource management problem. Then we focus on the Linux kernel, which is a widely adopted

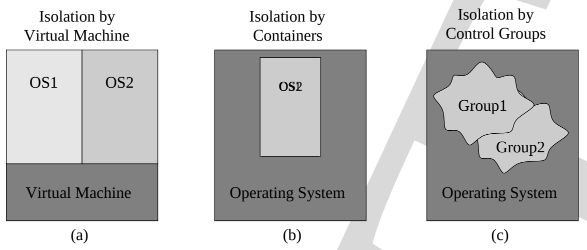


Fig. 6.6 The three different possible levels of resource control inside an OS.

operating system for a broad range of application contexts, and we describe the main mechanisms it offers to manage resources at run-time.

6.3.1 System-wideness and the OS

The operating system support for resource management has changed significantly in the course of the last decades. One can find in existing literature many studies on resource control at operating system level started. Systems were mainly based on main-frames, which had a central computation system and many users accessing it remotely and competing to the usage of really limited resources. Thus, the basic approach of dividing the available resources into several groups corresponding to different user profiles, e.g., gold, silver and iron, was pretty enough. For example, resources were always granted to ”gold users” and temporary given to ”silver users” once not needed by the first class. This simple approach allows to maximize the usage of the limited resources without any adverse impact on the expected performance of different users.

Nowadays we have multi-processor systems characterized with multiple CPUs and high bandwidth network (that too getting faster and faster) and with cheaper memories. These systems have been adopted initially on data centers, which use clusters of computation nodes, and the focus has moved on security and web based distributed access. This application scenario raised the interest for the virtualization support. Indeed, the business idea is to partition the resources of a big system (i.e., guest system) into smaller pieces (i.e., host systems), each one dedicated to different customers which actually pay for what they get in terms of resources. Thus, the new requirement has become the possibility to run in isolation multiple virtual systems on the same hardware resources.

The requirement to control the resources of a big system, in order to partition them in smaller pieces, has been supported by three different design approaches (Figure 6.6), which rely on: virtual machine, OS containers or flexible resource control. The *virtual machine* approach allows to run even a completely different OS on the same hardware platform [15]. While this approach grants the maximum security [27], thanks to the complete isolation of the host systems, it is also the

Table 6.1 The three different design approaches for resource management

	Virtual Machine	Container	RC
Performance	Not good	Very good	Good
Isolation/Security	Very good	Good	Not good
Runtime Flexibility	Not good	Good	Very good
Maintenance	Not good	Good	Good

one with higher maintenance efforts and run-time overheads [9]. Moreover, running hosts on complete isolation reduces also the run-time flexibility since it is practically impossible to share unused resources with overloaded hosts. The *container* based is a better approach for the performance and maintenance perspective. In this case the isolation is provided by a single OS that has complete control over all the available resources and gives each host system a partitioned view of them [12]. The resource control provided by a single OS allows better performance and a more flexible run-time resource allocation, thus maximizing their usage.

Being mainly addressed to resource isolation and security, these two approaches are interesting for application contexts in which we are interested into running different customers on the same hardware resources. This is the main case of web services, where multiple customers, even belonging to different companies, buy the usage of portions of the same physical resource. Of course this is not the only application scenario. In many other scenarios, we are interested in optimizing the usage of available resources by many tasks of the same system. This is the main case of mobile embedded devices, such as smartphones and set-top-boxes. In these application contexts we have a single user which runs multiple tasks corresponding to different workloads, either critical or best-effort. In these scenarios, the entities are tasks, instead of users, which compete to the usage of shared resources, instead of buying them. The resource management problem, in this specific scenario, can be efficiently solved using more flexible resource control systems. This last approach is still based on a manager running on a single OS which groups the available resources and then maps tasks on these groups according to some usage policy.

The main properties of the three different approaches are summarized in Table 6.1. In the next subsection we concentrate on the third approach to investigate deeply the support that a modern Linux kernel provides for the flexible control of resources.

6.3.2 OS mechanisms supporting RTRM

Almost all modern OSs provide resource management mechanisms to properly grant shared resources access to multiple tasks. Linux is perhaps the OS that has developed the most advanced mechanisms to support run-time resource management for a great number of resources. This is thanks to its wide adoption, on systems that range from high-end servers of data centers down to multimedia mobile systems of

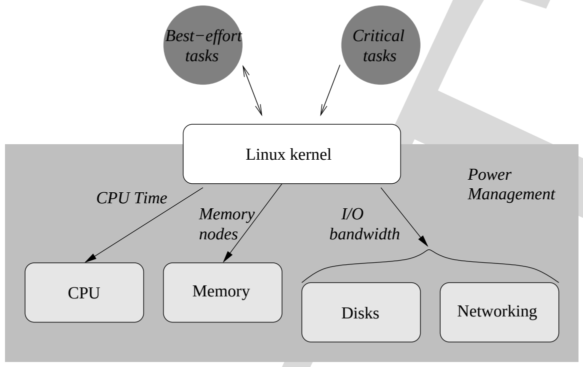


Fig. 6.7 Linux provides RTRM support for different subsystems. Tasks can be grouped into classes which correspond to different resources access priorities.

smartphones. As depicted in Figure 6.7 many Linux subsystems provide a resource manager, most notably: CPU, memory and the I/O communication channels such as disks and networks interfaces. Moreover, since power and energy consumption are critical aspects, for both high-end servers and mobile devices, the power management subsystem works parallel to the previous ones to manage properly electrical consumptions.

Past proposals on resource management and task grouping, were based on a basic abstraction that allows to group together multiple processes in order to track and limit the resources that they are allowed to access. To fulfill all these functionalities, the Task Control Groups (TCG) framework is available since Linux kernel version 2.6.24, when it was first developed and merged by Paul Menage of Google Inc.

When a task is associated to a particular *cgroup* (i.e., a control group), it will get an access to a portion of the system resources, where we can specify how big or small that share can be. These shares represent minimum values, not maximum. Thus, if we give one group 10% of a resource and another 90%, then if the more privileged group isn't using its full 90%, the other group can have whatever is left over. This borrowing mechanism allows to optimize resource usage while still granting each group the resources defined at design time whenever required.
This low-level mechanism was added in Linux mainly for containers and virtual machines. However, they are not restricted for that solely purpose. One of the most interesting approach nowadays is to employ it to manage resources—and therefore performance—of ordinary processes in a multi-tasking single user system. The idea is that other subsystems hook into the generic cgroup support to provide new attributes for them. For example, this allows to account or limit the resources that the tasks in a control group could access.
Like many other Linux frameworks, the TCG interface is organized as a virtual file-system, which can be used both to inquire about resources partitioning and to configure it by simply reading and writing files.

CPU Time management

The CPU subsystem allows to manage the CPU time resource. Different cgroups can be created and assigned to. The client of this subsystem is the Linux scheduler, which recalculates the percentage of the total CPU each cgroup will get. For example, we might create a cgroup for the background processes, another for the logged-in users and a third for root and daemons. If we gave them each one share, they would get a guarantee of no less than 33% of the CPU time.

Memory management

The next precious resource that can be managed is memory. The `memory` container allows to put a memory limit on a cgroup, and if the group's Resident Set Size (RSS) exceeds that number, its least-recently-used pages are swapped out.
This is definitely excellent for managing tasks with memory leaks, thus improving system stability. Moreover this can also be used for less severe problems. For example, tasks which use a lot of memory tend to push out pages of other tasks which haven't run lately, making them slow in restart. By putting a memory limit on the cgroup of memory, hungry tasks will grant a more rapid restart of other tasks.

Unfortunately, at least at the moment of this writing, the cgroups memory limit is a hard limit, and doesn't allow overcommitment even when we have lots of free memory. Thus, a natural extension to this container is the support for a soft limit, which should guarantee an upper bound like CPU share does. Such a support should allows a larger soft limit when there is plenty of free memory, but it reduces the limit to the smaller hard ones when there is demand for memory.

I/O management

A certain kind of abrupt slowdown is common to anything that can build up a queue, such as a CPU, a disk or a network, as well as any program that uses these resources. The queue network theory [13] shows that any open system with a service center s that require certain service time t_s to process a job exhibits performances which have the behavior represented in Figure 6.8. This is the classic behavior of a disk, where for example it can deliver 425 I/Os per second at 100% utilization. Thus, since we can't exceed 100% utilization, if we ask for 500 I/Os per second, only 425 are served while seventy-five disk access requests have to sit in a queue and wait.

To keep a devices in the "good" part of their response curves is an excellent reason to use resource limits, thus avoiding slowdowns. Four hundred requests at 40 milliseconds is far better than five hundred averaging more than 80 milliseconds each. Some experiments could give a rough estimate of the limit of a device, and we can use this value to set a limit that keeps them from being driven into overload. The solution to this problem is completed by limiting the I/Os that a task is allowed to issue. Thanks to its I/O scheduler, which is the natural client of the block subsystem,

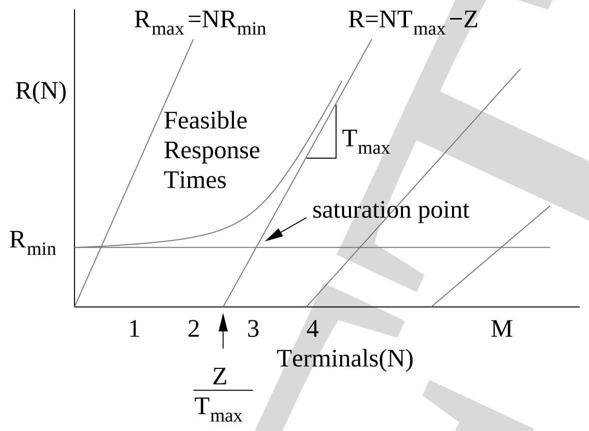


Fig. 6.8 The asymptotic system response time of an open system.

Linux shines on the management of resource such as disks and other input/output devices. The I/O scheduler has been extended to understand cgroups, and can refrain from dispatching I/Os if they will exceed the cgroup bandwidth ration. As usual, using the virtual file-system interface, the `blockio` subsystem allows to define a [MB/s] bandwidth limit of each char or block device node, by simply echoing that value within the provided `blockio.bandwidth` attribute.

Network management

The control of the network bandwidth resource is straightforward using the cgroup framework, but slightly different with respect to the previous subsystems. As in case of previous subsystems, a specific subsystem is provided for the resources, i.e. `tc`, which can be used to define classes of network traffic. However, the control groups defined are used to associate a “class id” to every network packet generated by tasks belonging to the corresponding group. The actual bandwidth control is in charge of the Linux traffic shaping framework, which is the client for this control group subsystem. A command line tool allows to associate bandwidth constraints to each class id defined by control groups.

General conclusion on resource management

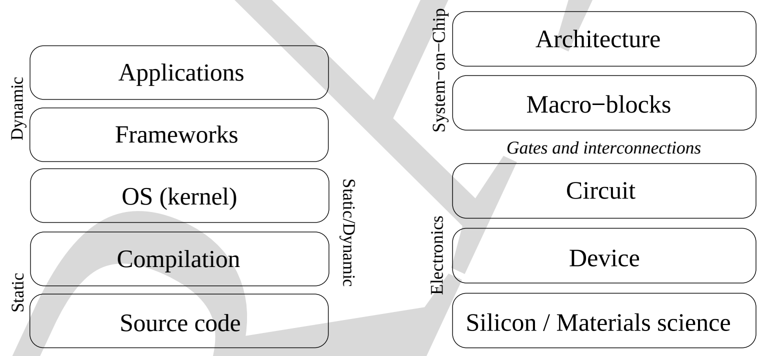
The first observation is about usability of the task control group framework: resource shares are not easy to reason about. Task control group framework only guarantees that when the machine is loaded you get a specific resources share. However, when the system is not loaded a task could get more, but how much is probabilistic. It depends on everything else that is running on the machine, and at the end this can

be confusing and also hard to handle for some tasks.
Because of the hard understanding of their behaviors, one could be tempted to only use hard limits. Hard limits are trivial to reason about, and they are genuinely useful for preventing catastrophes, such as a disk driven into infinite slowdown. However, hard limits should not be used as a general tool: doing so means that the tasks can never use any spare cycles that a system has. Instead any such spare cycles will be wastes, which is the same as wasting resources.

6.4 Power management

Digital electronics gives enough opportunities to reduce power consumption at different abstraction levels, not only through silicon physical optimization [24, 25]. As a matter of fact, power reduction opportunity increases with higher levels of abstractions, so that from architectural up to system software layer we have enough room to address the power/performance challenging trade-off.

The involved abstraction levels are shown in Figure 6.9. We have to decouple the SoC design in two planes: the software plane and the hardware plane. The former, shown in Figure 6.9(a), relates to the different levels at which the software can operate to effectively give contribution to power consumption reduction. The latter (Figure 6.9(b)), on the other hand, refers to the design of the underlying hardware, providing mechanisms to the upper levels of details.



(a) Software design and abstraction levels

(b) Hardware design and abstraction levels

Fig. 6.9 Power reduction and optimization techniques cover a wide range of SoC design, from both the software and hardware planes. An holistic low-power design methodology, where applicable, should consider crossing different abstraction levels for efficient and proficient power management and optimization.

From a software perspective, power reduction techniques can be employed both statically and dynamically. Static strategies are generally addressed at compile-time [22, 8], or at least through ad-hoc software architecture techniques at source-code level. Static techniques are of great importance since they can be used to exploit as much as possible the required power/performance requirements, but these techniques lack flexibility. This is furthermore true in those systems where the workload is not known in advance. For this scenario, dynamic approaches should be employed, for instance at the OS level (kernel) or at higher abstraction layers [3]. Applications can directly impact on the power/performance trade-off, but a more sophisticated mechanism can reside at the kernel level, where the OS is aware of the entire system status. The most complete software frameworks for the Linux kernel are reviewed in the next sections.

From a lower level perspective, hardware has to provide the software with control and observation points in order to ensure that the desired goal is achieved. Thus, the power/performance trade-off solution is searched in a hardware/software co-design approach, as it has been previously stressed in Section 6.2. While classical low-power design methodologies defined mechanisms to solve power issues from the physical up to the gate and architectural levels of abstraction, such methodologies are generally based on a precise hardware support, e.g. level shifters, PLL registers for clock signal [10]. In parallel, there are several software frameworks that address power management. Hereby, we will focus on those designed for Linux-based systems, and which were originally designed for general purpose platforms. Nevertheless, their applicability is of (quite) general validity, also for mobile embedded systems.

The available approaches can be conveniently grouped into two categories: pure-OS and cross-layer. The distinction comes from the power optimization mechanisms that are applied, and which kind of interaction is exposed to higher levels. A brief summary of the presented approaches is given in Table 6.2.

Table 6.2 This table shows a summary comparison among the presented software frameworks. The classification is made considering different aspects: static or dynamic power consumption involved, clock gating and power gating.

		Power Optimization		**Clock gating**	**MVS**	**Power gating**
		Static	Dynamic			
PURE OS	`CPUFreq`		●		●	
	`CPUIdle`	●		●	●	●
	`S/R Fw`	●				●
	`Clock Fw`		●	●		
	`V/I Fw`		●		●	●
CROSS-LAYER	Centralized (`DPM`)	●	●	●	●	●
	Distributed (`QoS`)	●	●			
	Hierarchical (`CPM`)	●	●			

The table reports the proposed classification in terms of pure-OS and cross-layer, and for each entry a comparison is performed against the power optimization and power optimization mechanisms involved: static versus dynamic, and clock gating versus power gating or voltage selection. A bullet (ﬄ) suggests that the current entry addresses that specific optimization, or supports that specific mechanism. Table 6.2 considers only three mechanisms for power management. Clock gating technique aims at reducing the dynamic power consumption by disabling (i.e., gating) the input clock signal. The frequency of the clock signal drops to zero, so that the switching activity drops to zero, too. Multi Voltage Scalings (MVS) refers to the use of different power rails, providing different regions of the chip with an ad-hoc voltage supply value. Last, power gating is the technique that cuts input voltage source, so that reducing quadratically the consumption of dynamic power.

6.4.1 Pure-OS Techniques

Pure-OS techniques are completely implemented within the Operating System; they do not provide support for direct input from applications. They attempt to figure out application requirements based on previously monitored behavior or current activity, and enforce some control decision either on a single device or on an entire subsystem. We can further divide these techniques in two groups, whether they tend to optimize static or dynamic power consumption. In the former case, namely *resource hibernation*, they are generally based on the exploitation of ON/OFF states of the peripherals. In the second case we refer to *resource tuning* techniques, since power minimization in obtained by properly configuring available operational parameters of the target platform, according to the changing run-time requirements. We have to further distinguish between device-specific techniques and system-wide techniques. The former class relates to those techniques addressing specific devices, while the latter attempts at optimizing the system as a whole, in a more abstract view of the application.

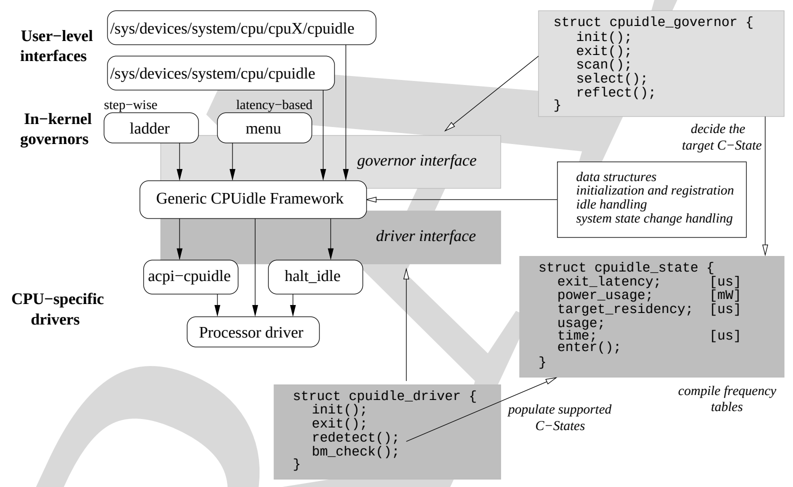
6.4.1.1 Device specific techniques

Being one of the more power demanding device, power optimization of the system processor is of major interest. There are two main frameworks available in a modern Linux kernel: one is devoted to reduction of static power consumption while the other addresses the optimization of dynamic power consumptions.

CPUIdle

The `CPUidle` framework [18] focuses on power management of an idle CPU. We refer to a CPU as being *idle* when it is doing nothing useful for the application itself, there is no workload, and hence it can be turned off to prevent from unnecessary power consumption. We have several opportunities in this context, ranging from clock gating or shutting down increasing portions of the circuitry, down to completely power gating the processor. These different solutions correspond to a well-defined set of idle states that modern high-performance processors exhibit. Idle states are characterized by particular processor configurations, with precise power consumption levels and wake-up latency. Moving from the simple approach of clock gating to power gating, there are increasing penalties, mostly related to wake-up latency. For instance, waking-up from an idle state requires just to re-enable the clock, while waking-up from a deep idle state could require to re-initialize the CPU and restore its registers from main memory too.

`CPUidle` addresses power/performance trade-off from a software layer standpoint, with the aim of exploiting all the available idle states of a processor without impacting on the overall system performance. An effective solution to this problem requires an adequate support to identify the real system requirements in terms of CPU latency. The current Linux implementation defines a proper software design which separates the low-level software mechanism from high level interface toward the framework clients to simplify this. An overall view of the framework architecture is given in Figure 6.10.



```
struct cpuidle_governor {
   init();
   exit();
   scan();
   select();
   reflect();
}
```

```
struct cpuidle_state {
   exit_latency;      [us]
   power_usage;       [mW]
   target_residency;  [us]
   usage;
   time;              [us]
   enter();
}
```

```
struct cpuidle_driver {
   init();
   exit();
   redetect();
   bm_check();
}
```

Fig. 6.10 A simple representation of the `CPUidle` framework.

The low-level interface [19] supports the definition and registration of processor-specific drivers. Those drivers are required to define the set of idle states available on the target CPU. Each state must be characterized by a set of attributes defining their power contribution, exit latency and a target residency time which is considered as necessary to get an advantage from entering that state. Every idle state could also be associated to a specific callback function which implements all the required low-level code needed to actually enter the state.

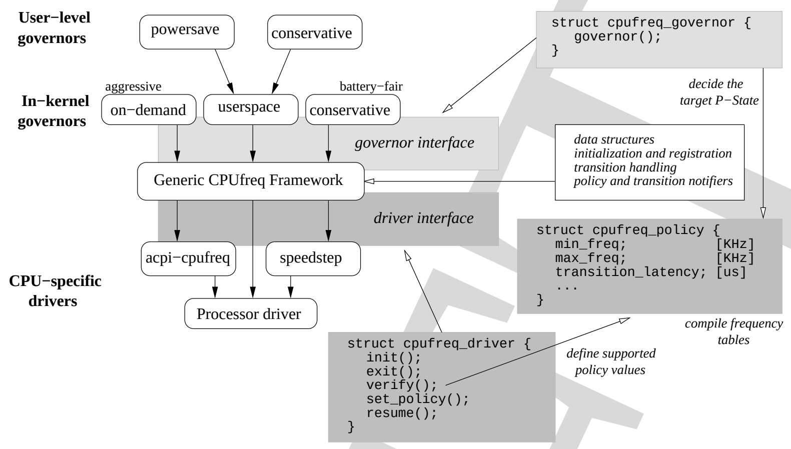
The high-level interface provides support for the definition of a *governor*, a processor-independent algorithm for choosing the effective idle state to enter, according to system constraints on maximum latency. There might be more than one governors registered in the core, but just one can be used at any time. Widely used implementations provide two governors, called *ladder* and *menu*. The ladder governor adopts a step-wide policy: every time the CPU is idle, a deeper idle state is entered only if we were previously able to remain in that state for a period greater than its corresponding target residency. Instead of relying on a simple heuristic approach, the policy implemented by the menu governor is latency-based. This policy exploits the information on the maximum allowed system latency in order to identify the idle state that should be reached every time there is an opportunity. This governor is certainly more efficient but requires a closer collaboration among applications and kernel drivers, to collect such requirements.

The core implementation is completely platform independent and provides the glue code that defines the required data structures, support drivers, governors registration and run-time selection. A proper monitoring interface is also exported to the collecting statistics on idle states usage.

CPUFreq

`CPUfreq` [20] focuses on the optimization of dynamic power consumption by exploiting DVFS mechanisms. A processor is in an *active state* when there is some workload ready to be executed. A workload can either be CPU-bounded or I/O-bounded; the former requires intensive CPU computations on memory located data, while the latter presents a more heavy information exchange toward relatively slow peripherals such as disks or low-bandwidth buses. In general, a single task cannot be exclusively classified in a single class; it happens that some portions are more CPU-intensive, while others are more like I/O operations. This means that the nature of a task could change during its execution; the combination of different workloads is even more evident if we consider a multi-tasking system with many concurrent applications running at the same time and sharing the few available processors.

The CPUfreq framework considers these combined behaviors in order to optimize power against performance. The basic idea is to exploit the possibility to perform computations at different operative frequencies. The set of available frequencies define the *performance states* of the platform; lower frequencies correspond to lower voltages and thus also less performing states with reduced power consumption and increased execution time. Switching from one performance state to an another one inserts an overhead that must be kept into consideration. Moreover, there is a need to identify efficiently the real system performance requirements. These observations make the CPU frequency scaling a rather complex mechanism to exploit. The framework available in Linux simplifies the implementation by a proper software design which aims at decoupling low-level software mechanisms from high level policies. An overall view of the software architecture is depicted in Figure 6.11.



```
struct cpufreq_governor {
   governor();
}
```

```
struct cpufreq_policy {
   min_freq;           [KHz]
   max_freq;           [KHz]
   transition_latency; [us]
   ...
}
```

```
struct cpufreq_driver {
   init();
   exit();
   verify();
   set_policy();
   resume();
}
```

Fig. 6.11 A simple representation of the `CPUfreq` framework software design.

The low-level software mechanisms [11] are implemented by *drivers*, required to define both platform specific information, and a set of control routines. The required information is related to the available performance state and the corresponding transition overheads, while the platform specific hardware mechanism to actually perform a transition must be wrapped by a set of properly defined callback functions. An high-level interface allows to define a *governor*, which is the platform independent algorithm for the evaluation of system performance requirements and of the selection of the optimal performance state. At least one governor must be defined, and multiple governors enable adaptive and dynamic multiple optimization strategies. The default framework implementation provides five governors, the more interesting and widely used being the *on-demand governor*. It implements a scaling policy based on the *run-to-idle* optimization. The CPU load is monitored in a periodic time frame, and according to the load observed in the past time frame a scaling decision is taken according to a simple rule: try to keep the CPU utilization around the 80% [20]. On CPU utilization higher that that threshold, an immediate scaling up to the maximum available frequency is required, to the contrary, on lower CPU utilization the scaling down is required step-by-step but only after a preconfigured number of negligible load time-frames are elapsed.

The *core implementation* provides the code to bind the platform independent governors down to the architecture specific driver. Moreover, a proper notification API is provided which allows other kernel components not only to be aware about CPU scaling operations but also to somehow interact with those optimization decisions, for example to assert a *veto* on some changes due to some contingent constraints.

6.4.1.2 System-wide techniques

The clock distribution tree and the power domains have some common characteristics: they have system-wide view, i.e., they interact with all the available on-chip devices, and they define a hierarchical dependency tree, i.e., a local power optimization decision could impact on different devices. These two components require system-wide optimization techniques which are able to collect information from multiple devices in order to identify a proper optimization strategy.

Suspend/Resume

The *Suspend/Resume Framework* (`S/R`) provides the proper support for a complete and efficient resource hibernation strategy. Linux supports three static-power saving states: *standby*, *suspend-to-RAM*, and *hibernation*. The main difference between them stands on how the device state is preserved. In a standby state a device is not functional, but it is still powered at least to grant the preservation of the content of its configuration registers. This kind of power saving addresses static power optimization, since the device logic is powered down and only a retention voltage is applied to the configuration array. This state could be always entered whenever a device is not in use since the recovery time is relatively short and practically negligible if compared to the typical operating system reaction time. In suspend-to-RAM a device is completely powered off, the contents of the configuration array are moved backed up in a secure area in main memory. Recovering from such a state is more time demanding since all the peripheral configurations must be recovered from main memory, and sometime this is possible only after a proper cold-start device initialization procedure. *Hibernation* is the more effective saving state: power consumption minimization is at its optimal value, saving the system configuration in a persistent storage and powering off all devices (memory included in some cases). Unfortunately, as one can argue, this last state is also the most expensive in terms of recovery time. A complete system restart is generally required, and it is done during the boot-up procedure in order to keep dynamic overhead at a minimum.

The main challenge for a successful implementation is the proper tracking of device functional dependencies. Different devices within a system could be interconnected to form subsystems. For instance a USB device, such as a memory stick, is connected to the port of an HUB which in turn connects to a port of a USB host controller. All this chain define a USB subsystem. Finally the host controlled could be either a system device or a gateway towards a PCI bus; which in turn defines another subsystem. Considering all the devices within a system and their inter-dependencies with respect to their functional dependencies what we get is logical dependency tree rooted at the CPU and having a device at each end node. This tree specifies an implicit order that must be respected both on suspend, starting the suspension from nodes and visiting the tree up to the root, and on resume, by converse visiting it starting from the root node down to the device nodes.

Clock framework

The `Clock Framework` has been introduced in the Linux kernel to optimize the dynamic power consumption associated to the clock distribution tree. The proposed framework is based on the management of the system clock signal. The hierarchical generation and distribution of the clock signal opens several opportunities for engineers to reduce power. The effective validity of the approach is driven by the fan-out value and the switching activity of the clock signal.

Purpose of the framework is to export the programmability of such components to the software level [26]. In this way, it is possible to cut-off some tree edges according to the desired computational activity; this is actively done by switching off a selected subset of LDO, PLLs or DIV modules. The approach takes even more advantage in some partitioned systems, in which several independent subsystems receive the clock from a common source, the top-level system clock, and scale the input signal according to local optimization policies, using DIV modules. In this way, individual operating requirements can be locally addressed with little silicon cost.

There are two main mechanisms for clock management: *clock stopping* and *clock scaling*. The former technique allows to disconnect the clock line from the associated PLL, and to eventually power off the PLL. Such mechanism gates the clock to the entire sub-tree controlled by the actual PLL that has been turned off. Clock scaling, on the other hand, does not disable clocks, but it instead scales down the incoming signal using physical dividers or reprogramming the top PLL for the current sub-tree.

Voltage and current control framework

The *Voltage and Current Control Framework* (`V/I Framework`) [6] is a quite specific support focusing on the efficiency of voltage regulators. In the architecture exploration, we highlighted how modern SoC architectures are composed of multiple voltage domains to fit specific requirements of each hardware block. In general, the voltage domains within a SoC could have some dependency relation between them. Sometimes these voltage domains are directly controlled by a dedicated voltage regulator usually provided by an external companion chip.

Each device in the system is powered by a certain voltage domain and, according to the specific functionalities required by a device, the current drained from the domain could also be very different. For instance, if we consider an audio-codec controller, its current drain is very different if we are listening to some audio stream via a loudspeaker or we are simply performing some digital audio mixing activities. A physics study of the dynamics of a regulator device shows that its efficiency is highly affected by the instantaneous current load. The *Regulator Power Efficiency (RPE)* of a regulator is defined in Equation 6.1.

$$RPE = P_{out}/P_{in} \tag{6.1}$$

Equation 6.1 compares the amount of power P_{in} that is presented as input to the regulator, and how much P_{out} we are able to derive from it; it is a direct measure of how much energy is lost in the regulator itself.

It is known that when the regulator works in normal mode, it is able to efficiently support only current loads over a certain threshold value. On the contrary, once the current load on the corresponding voltage domain drops under this threshold, the current requirement could be satisfied with a better efficiency only switching the regulator to an idle operating mode. This kind of behavior of voltage regulators are worth to be considered in order to implement a really holistic approach to power management in a modern embedded system. The framework presented in this paragraph has been introduced in the Linux framework quite recently, but provides a well designed and mature support to simplify the exploitation of this kind of optimization. The framework is composed of four separate interfaces:

- *regulator*, allows a regulator driver to register a set of required operations to the core framework;
- *consumer*, allows a device to notify voltage and current requirements to the regulator driver;
- *platform*, allows the system platform code to define the voltage domains, their dependencies and thus the creation of the regulator tree;
- *userspace*, exports a lot of useful voltage/current data and operation mode statistics via a `sysfs` interface to support device power consumption and status monitoring.

6.4.2 Cross-layer Techniques

Mechanisms and techniques supporting power management can be implemented at different abstraction levels; not only at architectural level but also at software level. Applications are aware of their Quality-of-Service (QoS) expectations. For instance, if we consider the playback of a network video stream: then we could easily identify at the application level some of the requirements, e.g., in terms of network bandwidth and decoding processing workload. Thus, the development of holistic approaches should support the aggregation of data from multiple layers into power management decisions. Cross-Layer techniques try to exploit mechanisms from different abstraction levels at the same time. The idea behind them is to provide properly defined mechanisms to collect abstract information from the higher abstraction levels, i.e., user-space applications, and exploit them to give some useful hints to the lower abstraction level techniques in order to improve the exploitation of the available architectural mechanisms. Cross-layer techniques aggregate data from multiple layers into power management decisions; indeed a properly defined interface allows the user space to assert Quality-of-Service requirements and exploit these information to support system-wide optimization. These techniques could be further grouped into two categories, *centralized* and *distributed*. *Centralized techniques* have been developed mainly to support the power optimization of relatively simple and dedicated embedded systems (e.g. personal media player), but these have some scalability problems related to their complexity which impacts on the implementation effort. On the other hand, *distributed techniques* are designed to be more scalable to easily address much more complex architectures (e.g. new generation smart-phones).

The power optimization techniques proposed in this class are essentially based on the definition of a single coordination entity, which stands in between the user-space applications and the available architectural mechanisms. However, we could identify essentially two orthogonal approaches: centralized and distributed; the main difference is in the role of the coordination entity. In centralized approaches the coordination entity has a direct control on the available mechanisms which are used to perform power management according to a single and system-wide optimization policy driven by the requirements collected from user-space. Distributed approaches, instead, not only implement a lightweight, single, and system-wide optimization policy but also exploit many other devices and subsystem specific policies. The idea is to implement a distributed control model where user-space requirements are aggregated and used to feed some input to more specialized local controls.

Dynamic Power Management

The `Dynamic Power Management` software (and hardware, refer to architecture, presented in [4], is both an architectural and interface proposal for a centralized cross-layer technique targeting high-performance embedded systems. Purpose of this proposal is to exploit effective power management mechanisms from the architectural view-point and from the management view-point at the OS level. This framework is neither a DVFS algorithm, nor a power-aware OS and also not a mechanism such as ACPI. Its relevance comes from the integrated engineering that has been applied to provide an highly efficient power management solution. To this purpose, the framework architecture is based on few abstraction objects: operating points, task states and policies. Each one cooperates information for performance and power management purposes. An overall representation of these abstractions is depicted in Figure 6.12.

An *Operating Point (OP)* is the lowest level abstraction which encapsulates a mixture of physical and logical parameters, representing a power-related sensible characterization. Each OP is thus a specific set of $\langle parameter, value \rangle$ pairs corresponding to a precise system power/performance configuration. At any given instant of time, the system is allowed to execute in a specific OP. Examples of operating points for a processor [17], as specified for the PowerPC architecture, are: core voltage, CPU operating frequency, bus frequency, and memory timing. The designer is in charge of the choice and setting of the OPs, as many as required by the capabilities of the target platform and the desired complexity of the framework implementation. The framework allows also the definition of *congruence classes (CCs)* which are sets of OPs that could be considered to be equivalent from certain power/performance optimization strategy. A *task state (TS)* is the high-level abstraction

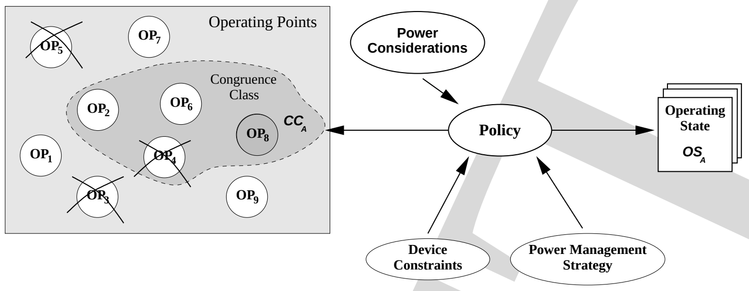


Fig. 6.12 The `DPM` architecture abstraction objects.

corresponding to a possible system operating state. In the control model defined by DPM, the system is seen as a state machine defined on a limited and well defined set of states. Example of states could be: idle, interrupt handling, CPU-bound process, I/O-bound process. The definition of the actual set of TS is once again in charge of the integration engineer. At run-time, each task could be associated with a task state. This mapping allows to identify in which task state the system is by simply looking at what task is scheduled to run at every time instant. Thus, switching from one task to another could imply the switching of the system among different task states.

Since each task state might have its own power/performance profiles, it is worth defining a mechanism to map task states to operating points, or more in general to a congruence class. This is achieved through the introduction of the *policy* abstraction, representing such mapping. According to the time of running task, the DPM core framework is able to automatically identify the current task state and accordingly map this on a congruence class defining a limited set of eligible operating points. Identifying a congruence class is not sufficient to actually select the best OP. To that purpose two more concepts are considered in the framework: the constraints and the optimization strategy. A *constraint* is a requirement on a specific OP value that could be asserted by either applications or device drivers. The core framework collects constraints asserted by all system entities and use them to invalidate the OPs that are not compatible with them. This first mechanism could thus reduce the number of eligible OPs available in the current congruence class. Finally, where more OPs that are still valid after considering all the constraints, the *optimization strategy* defines the ultimate rules to give each valid OP a relative preference value.

The framework is mostly an architectural proposal which requires customization efforts for each specific platform in order to be effectively used. The core framework provides just the *glue* code with the basic mechanisms to define the abstraction objects, but their actual definition is entirely an effort of the platform engineer. Secondly, the definition of the abstraction objects is a rather complex problem by itself since it requires a deep knowledge of a platform as a whole. Nevertheless, this remains one of the more interesting proposal for a centralized cross-layer optimization framework which is worth to consider especially in the case of relatively simple and dedicated embedded systems that require fine grained and low-overhead control.

QoS-PM Framework

The `QoS-PM Framework` has been the first attempt to implement a sufficiently generic framework to support distributed cross-layer power management within the Linux kernel. This kernel infrastructure has been proposed by Intel, essentially as an extension to the pre-existing *Latency Framework*, for optimizing the power consumption of a WiFi network interface [7].

The basic idea of this framework is to define a set of QoS parameters which are available to both applications and in-kernel code to assert requirements on them. The parameters defined are sufficiently abstract not to reduce the portability of the solution; in the current implementation they represent network throughput, network time-out and system latency. Of course this initial set of parameters is quite limited, but could be easily extended provided that the new parameters are completely platform independent. Well-defined and simple methods can be used to assert requirements on each of these parameters which are then aggregated by the core framework. The *requirements aggregation* is performed using a simple boundary function, i.e., the maximum or the minimum of the requests is considered to be the more restrictive value for the parameters. Drivers and other kernel code could declare their interest on a particular parameter by simply subscribing the corresponding notification chain. Once a new request on a parameter happens, the aggregated value is notified by the core framework to each driver or subsystem which has registered to the notification chain associated to that parameter.

Once a driver is notified by a new aggregated value for a certain parameter of interest, it could exploit that information in order to fine tune its local optimization policy. For instance, in the current implementation of the CPUidle framework described so far, when an idle state transition has to be decided, it takes into consideration the system latency requirements. This information is valuable since we know that the exit time from an idle state could be highly varying and thus could have also a great impact on the experienced latency of the system. The behavior implemented in this framework is thus that of a distributed control model. The framework core collects requests from applications and provides a simple optimization policy, based on the boundary aggregation, that deliver some tuning parameters to many other specialized policies. It is worth noticing that the current implementation of the notification mechanism support only a *best-effort* approach. Once a driver receives a notification of an update on a certain parameter, it could decide to do its best to satisfy the requirements. But if that is not possible, there is no feedback delivered up to the requesting user-space application.

The best-effort nature of the current implementation, along with the simple aggregation functions supported, are justified by the need to keep the QoS framework as simple as possible. By this design choice, the paradigm of a cross-layer distributed approach to power management is easily exported to the Linux kernel.

Nevertheless these are also some of the main limitations of the current implementation and motivate the research interest in this specific area of power management at operating system level.

Constrained Power Management

The `CPM framework` [2] is the first complete and comprehensive implementation of the *hierarchical power manager* concept for the Linux kernel. This framework is based on the design of a single coordination entity which allows both to exploit a system-wide view of resources availability and to aggregate all the application requirements.

Resource availability is defined by device drivers, in a platform independent way, and this information is exploited by the framework to support the system-wide optimization policy with fine-details and improved portability. The fine-details are granted by the low-level information collected at run-time by drivers, thus improving the portability of the control solution. By changing an architecture or even a single device, the new information is automatically detected.
Application requirements are collected by a single and well defined user-space interface. The framework aggregates all the requirements and translates them in a set of constraints for the global policy.
At run-time, a global optimization policy could exploit all the information collected either by drivers and applications: the former defining resource availability while the latter asserting QoS requirements. This information could be used effectively to solve a multi-objective optimization problem targeted to identify the best system-wide configuration. For scalability reasons, this configuration could not be completely defined by the coordination entity. Instead, this entity will notify proper constraints to drivers and let their local optimization policies to do the fine-tuning.

In order to efficiently support portability and scalability of the control policy, it has been defined on the base of abstraction and modeling layers, as depicted in Figure 6.13.
The *abstraction layer* grants portability without compromising the fine-details requirement. Available resources and devices working modes are represented in a platform independent way. Resources are abstracted using a set of metrics (PSM/ASM) which can be used also to setup the multi-objective optimization problem. A working mode of a device can also be represented in the space of these metrics by identifying a corresponding device working region (DWR).
The *model layer* exploit these abstraction information to automatically build a representation of all the system-wide feasible configurations (FSCs), each one identifying a working points of the entire system where a certain QoS level can be granted.
This model is suitable for supporting an efficient global optimization strategy provided by the *optimization layer*. This policy could support a multi-objective optimization strategy defined on the considered metrics.

The framework provides the implementation of a global optimization policy which rely on Linear Programming to identify a solution-equivalent and efficient

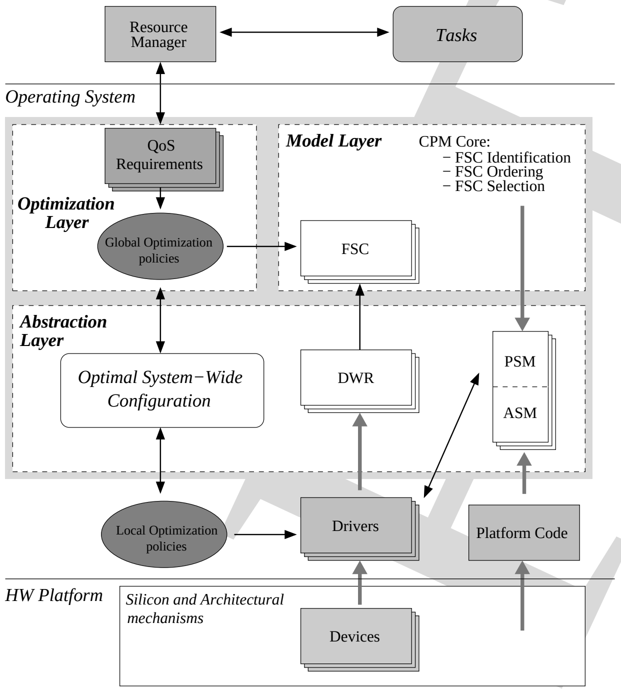


Fig. 6.13 The `CPM framework` architecture is based on three layers: abstraction, modeling and optimization.

optimization strategy. This strategy is based on three main tasks: FSC Identification, FSC Ordering and FSC Selection.
The *FSC Identification* exploits the information provided by the abstraction layer. This layer provides a multi-dimensional solution space, defined by the optimization metrics (PSM/ASM), and the device working regions (DWR) defined by each device. These information are used in the model layer to automatically identify the set of Feasible System-Wide Configurations (FSC) that identifies a platform independent representation of the system resources and capabilities.
In the *FSC Ordering* task, the global optimization policy exploits the FSC-based abstract system view provided by the model layer, to order the FSC previously identified according to the multi-objective optimization goal. Each time the system use-case change, the optimization goal is updated, and thus the FSC should be re-ordered.
Application requirements are collected and translated into constraints which invalidate unfeasible FSCs. The optimal FSC is selected, considering both the previously identified ordering and their validity defined by the current constraints, by running the *FSC Selection* task.
Each task has different run-time overhead and activation frequency. The FSC identification is the more complex task, it is required just at system-boot. Instead, FSC

selection must run each time an application assert a requirement but it has a negligible impact thanks to the support provided by the previous tasks.

6.5 Exploiting DSE to support RTRM

The Design Space Exploration (DSE) techniques have been demonstrated to be a valuable tool for the exploration and optimization of hardware platforms. However, their usage at run-time for the optimization of software behaviors has been only recently explored [14]. Even if the approach is promising, the integration of a DSE based optimization policy for the power optimization of a computing platform could exhibit less flexibilities. This is especially true if we consider the new generation systems, based on a deep sub-micron production processes, which run multiple applications on top of an Operating System.

This section highlight the potential shortcomings and outline a possible strategy for the integration of the DSE results with the Operating System and its Run-Time Resource and Power Management frameworks. At the time of this writing, this integration will be a part of future investigations, hence it is outside the scope of this book on MULTICUBE project.

6.5.1 Integration Pitfalls

There are two main obstacles to cope with at system-level: the possibility to have a mixed workload and the run-time phenomena. They are discussed below.

Mixed workload: Critical and Best-Effort applications

The modern mobile multimedia platforms are characterized by multiple applications, which can be either critical or best-effort. The former are applications, generally known at design time and provided by the device producer/integrator, which implement fundamental tasks for the target device. The latter instead are applications unknown at design time but that each user could add and use once the system is already in production.
The critical applications could be off-line fine tuned to be highly efficient, for example using DSE techniques, and their resource requirement are considered mandatory for a proper functioning of the device. On the contrary, best-effort applications could not be considered at design time, during the definition of the optimization strategy, but their run-time effects could have an impact on the overall system behavior that should be considered by the running optimization policy.

Run-Time phenomena

Let us consider a context of a platform where we admit the presence of multiple applications, running concurrently on the same shared resources, each one having its own *application-specific requirements*. In such a context, the efficiency of managing the available resources is a challenging goal. The mapping of applications onto the available resources may change during the device life-time, and the current effective mapping should be based on specific quantitative metrics, e.g., throughput, memory bandwidth and execution latency. In parallel to the application-specific requirements, we also experience other *non-functional requirements* such as power consumption, energy efficiency and thermal profiles.
The presence of these two types of requirements makes the mapping decision even more complex when we consider some characteristic run-time phenomena such as production process-variation, hot-spots generation, resource failures and workload fluctuation.

6.5.2 Integration Requirements and Goals

The idea to integrate effectively the DSE techniques within an Operating System cannot disregard the problems highlighted in the previous section. Moreover, to provide a general control solution which is acceptable by the Linux community, it is required to integrate the DSE control with the existing frameworks.
To satisfy all these goals, by providing a flexible and efficient OS integration of the DSE control techniques, a hierarchical design should be considered. In such a design, the monitoring, management, control and optimization strategy is operated at different granularity levels. Each granularity level collects requirements from higher level, runs a specific optimization policy, and finally identifies a set of constraints delivered to lower levels.

Such a hierarchical approach require the development of a new framework, in between the DSE control policy and the in-kernel existing frameworks, which allows to collect application requirements and match them against resources availability. This matching is performed based on a dynamic optimization policy to be developed at different abstraction levels and according to different optimization goals. Run-time optimization policies derived from DSE are considered as coarse grained configuration points, related to the specific off-line profiled application. Nevertheless, such information could be aggregated at run-time with other system requirements in order to achieve a fine-grained and system-wide optimization.
The main *abstraction levels* are user-space, kernel-space and device-space. The user-space level corresponds to the DSE control policy, which can be defined off-line for the fine tuning of profiled critical applications. An example of those kind of policies is presented in [14]. The kernel-level control policy is defined by the new framework, which allows to collect and aggregate requirements from both critical and best-effort applications. This level is somehow similar to the cross-layer

frameworks presented in Sec. 6.4.2. Finally, the device-space level is related to each pre-existing subsystem specific control policy like those presented in Sec. 6.4.1. Regarding the *optimization goals*, the definition of a new kernel-space framework, which could have a system-wide view on run-time system state and resource availability, allows more easily to develop a control policy which is aware of run-time phenomena. This new framework could provide support for optimization goals which are difficult to define off-line, such as resource usage fairness, application performance, power consumption and thermal management.

The new RTRM kernel-space framework proposed for the integration of the DSE defined control policies allows to effectively target two main goals: dynamic resource partitioning and resource abstraction.

Dynamic Resource Partitioning

Each application could be associated to a certain “priority level” related to the different impact that the application (either critical or best-effort) could have on the overall user experience. Thus, the RTRM framework should provide a support to handle both: critical workloads, that have hard requirements in terms of resource usage along with execution behaviors, and best-effort workloads, for which a penalty either does not impact on perceived behaviors or produce a tolerated QoS degradation.

Both classes of applications should access the available resources through a single run-time resource manager framework. Thus, the role of this framework is to account for available resources, and grant access to these resources to demanding application according to their priority level. To efficiently manage this scenario, the resources could be dynamic partitioned, taking into consideration current QoS requirements and resources availability. This dynamic partitioning will allow to grant resources to critical workloads while dynamically yield these resources to best-effort workloads when (and only while) they are not required by critical ones, thus optimize resource usage and fairness.

Resource Abstraction

To suitably tackle the run-time phenomena problem, the RTRM kernel-space framework should handle a decoupled perspective of the resources between the users and the underlying hardware. The user applications will see virtual resources but they will not be aware of which of the physical resources are effectively available. At run-time the RTRM will perform the virtual-to-physical mapping according to the current objective function (low power, high performance, etc..) and run-time phenomena (process variation, temporal and spatial temperature gradients, hardware failures and, above all, workload variation).

6.6 Conclusions

The aim of this chapter is to provide the big picture of the components and goals of a system-wide resource manager. The need for a Run-Time Resource Manager(RTRM) is dictated by the non-deterministic nature of the complex modern applications. This need, in addition, poses new challenges in finding out a low-overhead solution that satisfies all the functional and non-functional requirements. From a performance and QoS view-point, having multiple applications means having at any point of time different perspectives of the entire system.

Our current research takes full advantages from the DSE-based design flow developed within the MULTICUBE project. The coarse grain analysis of the best operating points to be used at run-time will be improved and integrated in a more comprehensive framework, working at the level of the operating systems, capable to provide a fine-grain tuning of the system configuration considering a system-wide optimization perspective. Our research goal is to develop a hybrid centralized/distributed approach, conveying into a hierarchical strategy for managing power consumption/energy requirements. A hierarchical strategy has the benefits of gathering local control and centralized optimal solution to the problem. In addition, such a hierarchical control will be based on a hardware/software co-design approach, where the solution is based on hardware and software components.

References

1. International Technology Roadmap for Semiconductors, 2009 Edition. Available at http://www.itrs.net
2. Bellasi, P., Fornaciari, W., Siorpaes, D.: A Hierarchical Distributed Control for Power and Performances Optimization of Embedded Systems, *Lecture Notes in Computer Science*, vol. 5974. Heidelberg, Springer Berlin, Berlin, Heidelberg (2010). DOI 10.1007/978-3-642-11950-7. URL `http://www.springerlink.com/content/8v77262826305r87`
3. Benini, L., Bogliolo, A., De Micheli, G.: A survey of design techniques for system-level dynamic power management, vol. 8. Kluwer Academic Publishers, Norwell, MA, USA (2002)
4. Brock, B., Rajamani, K.: Dynamic power management for embedded systems. Proceedings on IEEE International SoC Conference pp. 416–419 (2003). DOI 10.1109/SOC.2003.1241556
5. Cumming, P.: The TI OMAP Platform Approach to SoC, chap. 5. Kluwer Academic Publishers (2003)
6. Girdwood, L.: Every Microamp is Sacred - A Dynamic Voltage and Current Control Interface for the Linux Kernel. Tech. rep. (2008)
7. Gross, M.: The PM_QoS framework documentation. Proceedings of Linux Symposium (2008). URL `http://lxr.linux.no/linux+v2.6.34/Documentation/power/pm\_qos\_interface.txt`
8. Hsu, C.H., Kremer, U., Hsiao, M.: Compiler-directed dynamic voltage/frequency scheduling for energy reduction in microprocessors. In: ISLPED ’01: Proceedings of the 2001 international symposium on Low power electronics and design, pp. 275–278. ACM, New York, NY, USA (2001). URL `http://doi.acm.org/10.1145/383082.383165`
9. Karger, P.A., Safford, D.R.: I/O for Virtual Machine Monitors: Security and Performance Issues. IEEE Security & Privacy Magazine **6**(5), 16–23 (2008). DOI 10.1109/

MSP.2008.119. URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4639018

10. Keating, M., Flynn, D., Aitken, R., Gibbons, A., Shi, K.: Low Power Methodology Manual: For System-on-Chip Design. Springer Publishing Company, Incorporated (2007)
11. King, R.: The CPUfreq framework documentation (2001). URL http://lxr.linux.no/linux+v2.6.34/Documentation/cpu-freq/
12. Lakshmanan, K., Rajkumar, R.: Distributed Resource Kernels: OS Support for End-To-End Resource Isolation. IEEE (2008). DOI 10.1109/RTAS.2008.37. URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4550792
13. Lazowska, E.D., Zahorjan, J., Graham, G.S., Sevcik, K.C.: Quantitative system performance: computer system analysis using queueing network models. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1984)
14. Mariani, G., Avasare, P., Vanmeerbeeck, G., Ykman-Couvreur, C., Palermo, G., Silvano, C., Zaccaria, V.: An industrial design space exploration framework for supporting run-time resource management on multi-core systems. Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010 pp. 196–201. URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5457211
15. Murphy, M., Fenn, M., Goasguen, S.: Virtual Organization Clusters. 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing pp. 401–408 (2009). DOI 10.1109/PDP.2009.23. URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4912960
16. Nollet, V., Verkest, D., Corporaal, H.: A Safari Through the MPSoC Run-Time Management Jungle. Journal of Signal Processing Systems (2008). DOI 10.1007/s11265-008-0305-4. URL http://www.springerlink.com/content/t4346j5t13451016
17. Nowka, K.J., Carpenter, G.D., Brock, B.C.: The design and application of the PowerPC 405LP energy-efficient system-on-a-chip. IBM J. Res. Dev. **47**(5-6), 631–639 (2003)
18. Pallipadi, V.: cpuidle - Do nothing, efficiently... In: Proceedings of Linux Symposium (2007). URL http://ols.108.redhat.com/2007/Reprints/pallipadi-Reprint.pdf
19. Pallipadi, V.: The CPUidle framework (2007). URL http://lxr.linux.no/linux+v2.6.34/Documentation/cpuidle/
20. Pallipadi, V., Starikovskiy, A.: The ondemand governor: past, present and future. In: Proceedings of Linux Symposium, vol. 2, pp. 223-238 (2006). DOI http://ftp.kernel.org/pub/linux/kernel/people/lenb/acpi/doc/OLS2006-ondemand-paper.pdf
21. Paulin, P.: SoC platforms of the future: challenges and solutions. In: MPSoC Forum 2005 (2005)
22. Saputra, H., Kandemir, M., Vijaykrishnan, N., Irwin, M.J., Hu, J.S., Hsu, C.H., Kremer, U.: Energy-conscious compilation based on voltage scaling. In: LCTES/SCOPES ’02: Proceedings of the joint conference on Languages, compilers and tools for embedded systems, pp. 2–11. ACM, New York, NY, USA (2002). URL http://doi.acm.org/10.1145/513829.513832
23. Shenoy, P., Hasan, S., Kulkarni, P., Ramamritham, K.: Middleware versus native OS support: architectural considerations for supporting multimedia applications. IEEE Comput. Soc (2002). DOI 10.1109/RTTAS.2002.1137378. URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=1137378
24. Unsal, O.S., Koren, I.: System-level power-aware design techniques in real-time systems. Proceedings of the IEEE **91**(7), 1055–1069 (2003). DOI 10.1109/JPROC.2003.814617
25. Venkatachalam, V., Franz, M.: Power reduction techniques for microprocessor systems. ACM Comput. Surv. **37**(3), 195–237 (2005). DOI http://doi.acm.org/10.1145/1108956.1108957. URL http://portal.acm.org/citation.cfm?id=1108956.1108957\&coll=GUIDE\&dl=GUIDE\&CFID=48769080\&CFTOKEN=23809990
26. Yermalayeu, S., Vervoort, G., Mahadevan, S., Becking, B.: Linux clock management framework (2007)

27. Zhao, S., Chen, K., Zheng, W.: The Application of Virtual Machines on System Security. IEEE (2009). DOI 10.1109/ChinaGrid.2009.45. URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=5328990