POLITECNICO DI MILANO
Dipartimento di Elettronica e Informazione
Dottorato di Ricerca in Ingegneria dell'Informazione

Cross-Layer Frameworks for Constrained Power and Resources Management of Embedded Systems

Doctoral Dissertation of
Patrick BELLASI

Advisor
Prof. William FORNACIARI

Tutor
Prof.ssa Letizia TANCA

Supervisor of the Doctoral Program:
Prof. Patrizio COLANERI

2009 — XXII

This document was sent to the publisher on January 25th, 2010.

The author can be reached for comments, questions and suggestions at the following e-mail address: derkling@gmail.com.

to all my dears

Contents

Abstract

Resources management, and thus also energy saving, has become a high priority design goal for embedded multimedia mobile devices such as smartphones. Such devices are usually based on platforms using a System-on-Chip (SoC), which embeds a number of peripherals sharing some resources and competing on their usage. Each one of these many embedded devices are usually characterized by a set of working modes, each one corresponding to different profiles of power consumption and corresponding performances.

Modern consumer electronics products provide also multiple functionalities, ranging from classic phone calls to more complex use-cases which could involve network access and multimedia data processing such as audio-video decoding and playback. These multiple usage scenarios are usually characterized by a competition on the limited resources available and thus could also involve conflicting requirements on the underlying hardware. Therefore, a proper resources and power management of such new generation platforms has become a more and more complex added value.

The ultimate goal of such a management is the search of the optimal trade-off between power saving and performances perceived by the user. Thus, it is worth to treat it as an *optimization problem* and investigate on the definition of solutions that are easily portable among different products.

In the light of these considerations, to effectively support the optimization of a complex platform, it is necessary (i) to have an updated and system-wide view of the available resources, (ii) to collect and aggregate QoS requirements defined by the multiple running use-case scenarios and (iii) to exploit these information within a dynamic system-wide optimization policy.

This problem is not effectively solved by current approaches. A number of frameworks has been developed that focus on specific subsystems or even single devices. These approaches result in multiple optimization strategies being implemented within the same system, with the consequent risk of overlapping control actions and conflicting decisions. Thus, these specialized frameworks used alone cannot grant system-wide optimizations.

The user-space approaches that have been investigated solve this problem by means of a centralized resource and power manager which is implemented as a middleware between the applications and the operating system. Unfortunately, these approaches require to modify applications, which is not always possible. Moreover, being implemented outside the operating system they are to much abstract to efficiently exploit all the underlying hardware capabilities.
Finally, the cross-layer frameworks implemented in the past are too complex to be effectively used in production devices or too simple to produce interesting results.

In this work, I propose a novel methodology which is able to efficiently support both resources management and power optimizations without affecting user-perceived performances. With the proposed approach, we avoid to design a centralized system-wide controller and to completely re-design it once the architecture or some of its hardware components changes. Instead, a single global optimization policy is designed to provide a coarse-grained tuning for already existing low-level and device-specific optimization policies. Indeed, this policy is designed to find a system-wide solution to the power consumption vs perceived performances problem. The solution is configured as a set of constraints that are notified to the multiple underlying device-specific and fine-tuning policies.

This methodology leverage a hierarchical design where specific domain experts could concentrate only on the definition of local optimization policies, targeting single subsystems or devices, by exploiting run-time information coming from the global policy. This global optimization policies take care also of collecting and aggregating resources requirements from applications, thus providing a complete and efficient in-kernel implementation of the cross-layer approach.

A framework implementing the proposed methodology has been developed and integrated within the mainline Linux kernel. Its design has been defined both to be efficient and to simplify the integration of both new and existing local control policies. The experimental results show that our implementation exhibits negligible run-time overheads while allowing to track correctly resources usage and identifying opportunities for power optimizations that do not impact on perceived performances.

Estratto in lingua italiana

La gestione delle risorse, ed in particolare il risparmio energetico, sono fra gli aspetti critici nella progettazione di dispositivi embedded multimediali portatili come i telefoni cellulari di nuova generazione. Tali dispositivi sono di solito basati su piattaforme System-on-Chip (SoC), costituite da una serie di periferiche che condividono alcune risorse e competono per il loro utilizzo. Ciascuna di queste periferiche interne al SoC è generalmente caratterizzata da più modalità di lavoro, corrispondenti a diversi profili di consumo energetico e prestazioni.

Questi moderni dispositivi portatili supportano ormai un numero sempre maggiore di funzionalità, e non si limitano più alle classiche attività correlate alla gestione delle chiamate telefoniche. Scenari di utilizzo ben più complessi sono ormai all'ordine del giorno e possono coinvolgere l'accesso ad Internet e l'elaborazione di dati multimediali, come nel caso della riproduzione di contenuti audio e video. Questo tipo di applicazioni sono generalmente caratterizzate da una maggiore competizione per l'uso delle limitate risorse a disposizione, addirittura potrebbero anche implicare requisiti contrastanti sull'hardware sottostante. Pertanto, sebbene la gestione delle risorse e della potenza sia indubbiamente un valore aggiunto per i dispositivi mobili di nuova generazione, una sua corretta implementazione sta anche diventando sempre più complessa e richiede lo sviluppo di adeguate metodologie.

Alla luce di queste considerazioni, al fine di supportare efficacemente l'ottimizzazione di piattaforme complesse, è necessario (i) avere una visione complessiva ed aggiornata dello stato del sistema e delle risorse disponibili, (ii) raccogliere e aggregare opportunamente i requisiti di qualità di servizio definiti dai molteplici scenari d'uso in esecuzione, ed infine (iii) utilizzare queste informazioni all'interno di una opportuna politica di ottimizzazione globale del sistema. L'obiettivo finale del sistema di gestione sarà la ricerca del compromesso ottimale tra risparmio energetico e prestazioni percepite dall'utente. Si tratta quindi di un *problema di ottimizzazione* che richiede soluzioni che siano non solo efficaci ma anche facilmente adattabili a differenti ambiti applicativi e dispositivi.

Purtroppo, questo problema non è ancora stato risolto in modo soddisfacente.

Un certo numero di politiche di ottimizzazione sono già state sviluppate, principalmente concentrandosi su sottosistemi specifici o addirittura singole periferiche. Questi approcci hanno comportato l'impiego di strategie di ottimizzazione multiple all'interno di uno stesso sistema, con il conseguente rischio di sovrapposizione delle azioni di controllo o ancora peggio della possibilità di decisioni contrastanti e quindi potenziali instabilità del sistema. Altri approcci hanno cercato di affrontare il problema in spazio utente per lo più proponendo dei controllori centralizzati realizzati da uno strato software che si frappone fra le applicazioni ed il sistema operativo. Tali tecniche richiedono però generalmente un'opportuna modifica delle applicazioni, cosa non sempre possibile. Inoltre, essendo esterni al sistema operativo l'elevata astrazione rispetto alla piattaforma sottostante non consente di sfruttare appieno le capacità offerte dall'hardware. Infine, gli approcci così detti cross-layer implementati in passato sono troppo complessi per poter essere utilizzati proficuamente in dispositivi commerciali oppure troppo semplici per garantire risultati interessanti.

Questo lavoro propone una nuova metodologia in grado di supportare efficacemente il problema di ottimizzazione energetica e gestione delle risorse senza impattare negativamente sulle prestazioni percepite dall'utente. L'approccio proposto evita di dover progettare un nuovo sistema di controllo centralizzato ogni volta che si cambia piattaforma o anche un solo suo componente. Infatti, una singola politica di controllo centralizzata può essere progettata per interagire con pre-esistenti politiche di controllo locale di più basso livello fornendogli dei parametri di calibrazione. Infatti, tale politica centralizzata è definita per l'individuazione di una soluzione ottimale globale del problema di bilanciamento dei consumi energetici rispetto alle prestazioni percepite dall'utente. Tale soluzione si configura come un insieme di vincoli che possono essere notificati alle sottostanti politiche locali deputate invece al controllo fine dei singoli dispositivi.

La metodologia proposta si basa quindi su un approccio gerarchico al problema di controllo. In tal modo gli esperti di ciascun dispositivo possono concentrarsi sulla definizione delle relative politiche di ottimizzazione locale che possono però sfruttare durante la loro esecuzione le informazioni rese disponibili in modo trasparente dalla politica di ottimizzazione globale. La politica di ottimizzazione globale si occuperà inoltre di raccogliere ed aggregare opportunamente le richieste d'utilizzo delle risorse avanzate dalle applicazioni, realizzando quindi una completa ed efficiente implementazione in kernel-space degli approcci cross-layer.

Un framework che implementa la metodologia proposta è stato sviluppato ed integrato nell'ultima versione del kernel Linux. Tale implementazione è stata progettata in modo da essere efficiente e semplificare l'integrazione di politiche di controllo locali sia nuove che pre–esistenti. I risultati sperimentali mostrano che la nostra implementazione ha un trascurabile impatto sulle prestazioni del sistema in esecuzione, consente di tracciare correttamente l'utilizzo delle risorse ed anche di identificare correttamente le opportunità di ottimizzazione energetica che non impattano sulla prestazioni percepite dall'utente.

Chapter 1

Overview

“ Engineering is the art of directing the great sources of power in nature for the use and convenience of man. ”

Thomas Tredgold

This chapter provides an overview of this thesis, including its goal, its motivation, its fundamental approach and the benefits it offers. It first motivates why embedded systems’ designers need a new generation of system-wide power and performances optimization techniques, on the basis of the current context. From this context I will derive the requirements that such techniques must meet. Finally, I will show how these requirements derive precise research choices, leading to a specific technique, which I will adopt.

1.1 Designing embedded systems is getting more and more difficult

In the last fifty years we have assisted to a sustained growth in the ability of silicon manufacturers to fit more and more transistors in the same area, and to raise the clock frequency of their devices. The processing power made available by microprocessors and programmable devices grew accordingly. Gordon E. Moore was the first, in 1965 to recognize [1] that the transistor density was growing exponentially over the years, and to capture this observation in his famous “law”, originally formulated as: *«The complexity for minimum component costs has increased at a rate of* *roughly a factor of two per year»*[1].

When Moore's Law remained in lockstep with classical Dennard scaling [2], which predicts faster, lower-power transistors at each fabrication node, the architectural focus on big, fast processors made sense. However, Dennard scaling ceased to provide big gains in speed and big reductions in power dissipation starting at the 90nm node [3]. Since then, CMOS circuits continue to get smaller but they don't get faster or drop in power consumption nearly as fast as they did before IC lithographies hit 90nm. Consequently, power dissipation and energy consumption started to become unmanageable at this node [4]. The problem is getting worse with each new process node. Embedded systems' designers must now adopt design styles that reduce system clock rates if they are to meet power and energy consumption goals.

Since some year big semiconductor vendors already offer multicore, symmetric multiprocessing (SMP) processors [5]. Each SMP core can run multiple, concurrent applications' threads. Such multicore processors were found first in large servers and laptops to run applications based on the "SAMD" (single application, multiple data) model. Nowadays a lot of interest, excitement, and worries are stimulated by the application of those same SMP multicore architectures also to embedded designs [6], where at a first instance we could believe that only few applications are actually "embarrassingly parallel".

The current performance improvement trend uses parallelism to motivate the development of architectures that combine both fine grained and coarse grained parallelism in systems with tens or hundreds of processors. These systems can be considered "manycore" processor systems, with the goal of achieving higher parallel code performance [7]. Manycore processor systems have tremendous potential for high-performance computing and scientific applications. However, these architectures are going to be explored not only to be used as accelerators in the design of tera-flop or peta-flop computers, but also in the domain of mobile multimedia embedded system [8]. Indeed, the significant increase in parallelism within a processor can lead many benefits including higher power-efficiency and better memory latency tolerance.

[1]To read reprints of Gordon Moore's 1965 and 1975 papers along with recent commentaries on Moore's Law, see the September 2006 issue of the *IEEE Solid-State Circuits Society Newsletter*.

Asymmetric multicore architectures Multi-core technology is presently massively used on embedded systems addressed to the multimedia mobile market such as that of 'nettops' and 'smartphones'. In the embedded system world, by expanding architectural thinking beyond SMP multicore architectures, it was even before possible to uncover at least two kinds of easily used concurrency that exploit heterogeneous rather than homogeneous architectures.

The first sort of parallelism is referred as "compositional concurrency" where various subsystems are woven together into a product. Each subsystem contains one or more processors optimized for a particular set of tasks. Communications within this architectural design style are structured so that subsystems interact only when needed. Figure 1.1 shows a block diagram of a Super 3G mobile phone that illustrates this idea. Compositional concurrent design offers many advantages:

- distributing computing tasks over several on-chip processors trades additional transistors in exchange for a lower clock rate to reduce overall power and energy consumption.
- dedicated subsystems can be powered down when not needed;
- application-specific instruction set processors (ASIPs) that are much more area and power efficient than general-purpose processors can be designed for each task processor;
- avoids the complex interactions and the synchronizations required between subsystems that are frequently associated with SMP hardware designs and multithreaded code;

The second form of convenient concurrency is referred as "pipelined dataflow", which is possible every time the computation can be divided into a pipeline built from individual task engines. Each pipelined task engine accepts, processes, and then emits data blocks. Once a processing task completes, the processed data block passes to the next engine in the chain. Such asymmetric multiprocessing algorithms appear in many signal and image processing applications[2]. Pipelining permits substantial concurrent processing and also allows even sharper application of ASIP principles because each processor in the pipeline can be highly tuned to just one part of the task.

These two types of convenient concurrency complemented each other. By combining the compositional subsystem style of design with pipelined, asymmetric multiprocessing (AMP) in each subsystem made it possible to built products in the consumer, portable, and media spaces with thousands of processors. Since many concurrent algorithms can be individually accelerated using compositionally concurrent design, and because many subsections of a single algorithm can be accelerated using pipelined design, the system problem becomes many pieces of code, all individually open to acceleration. The overall benefit of using multicore design has

[2]e.g. from cell phone baseband processing to video and still image processing

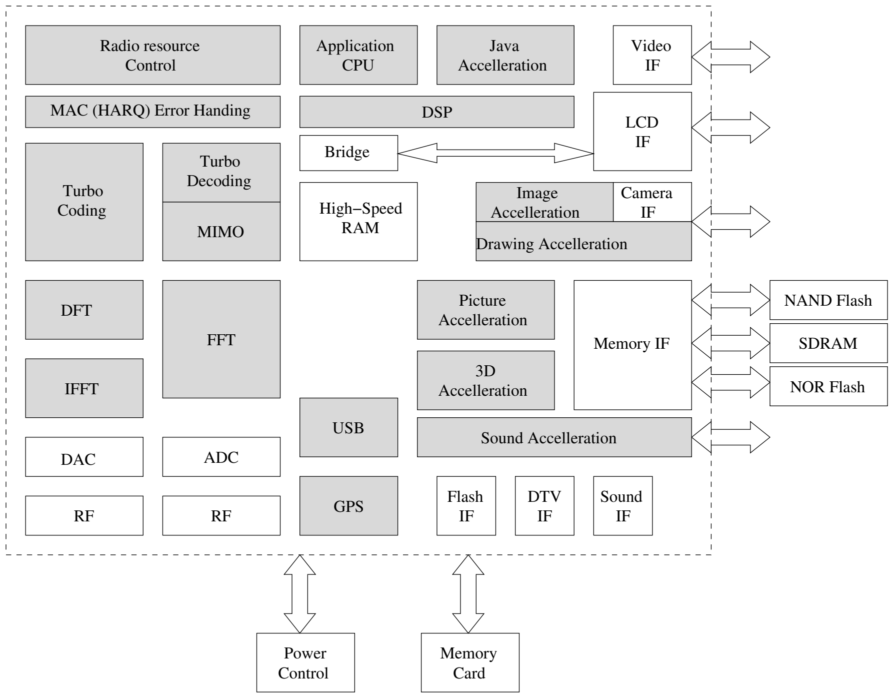
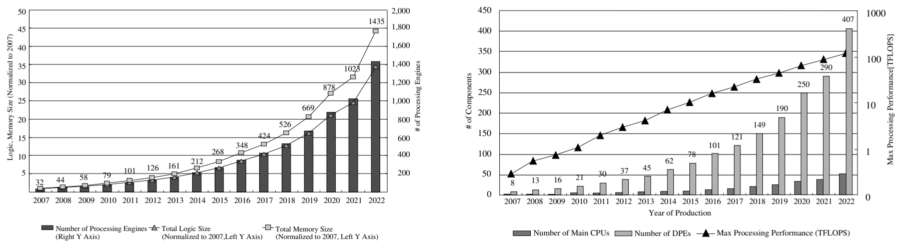


Figure 1.1: The block diagram of a Super 3G mobile phone. There are 18 identified processing blocks (shown in gray) in the figure, each with a clearly defined task. It's easy to see how one might use as many as 18 processors (or more for sub-task processing) to divide and conquer this system design.

been therefore much greater than Amdahl's Law predicts, while the requirements for software development do not substantially change.

Of course there was a downside for these benefits. The cost for cleanly separating and accelerating multiple algorithms is an increase in the number of transistors per design, as realized through the use of multiple processors. That has always been a fundamental cost of the "divide-and-conquer" design approach. However, Moore's Law, which trumps Amdahl's Law in this situation, has ensured that the cost of more transistors is very low – and likely advantageous – with respect to the much higher costs associated with high system energy consumption and development complexity arising from multithreaded software.



(a) Portable devices (b) Stationary devices

Figure 1.2: ITRS 2007 SoC consumer portable and stationary design complexity trends. Trend for the number of processing elements (PE's) predicted over the next 15 years in consumer portable devices, and the number of Data Processing Engines (DPE's) in consumer stationary devices.

Emerging symmetric architectures Programming AMP applications is far easier than programming multithreaded SMP applications because there are far fewer inter-task dependencies to worry about. Experience shows that it is possible to cleanly write software in this manner because many optimization issues arising from the use of multithreaded applications running on a limited set of identical processors are simply avoided. Therefore, by exploiting the two forms of "convenient concurrency" previously described it is possible to substantially free software developers from the need to think in terms of parallel operations because the various concurrent tasks are not so closely linked.

Nevertheless, even if is has granted increased performances in this few past years, asymmetric parallelism seem to be no more sufficient to support the requirements for the more and more advanced applications of next generation devices. The International Technology Road-map for Semiconductors [4] has estimated, over the next 15 years, that the number of processing elements (PE's) will grow up to 1435 in consumer portable devices (Figure 1.2a), such as mobile phones with extensive media capabilities or digital cameras. According to the same report, the number of Data Processing Engines (DPE's) in consumer stationary devices, such as high end game playing machines, will grow up to 407 with one main CPU every 8 DPEs (Figure 1.2b).

To feature hundreds or even thousands of processing cores on a single-die many-core processor, the challenges of energy consumption and performance scalability must be addressed within a given die area budget. Current commercial designs focus on MIMD-style multicores built with rather complex cores. While such designs provide a degree of generality, they may not be the most efficient way to build processors for applications with inherently scalable parallelism.

Increased power consumption threats One of the main threat to the feasibility of products exploiting the new technologies and applications presented above is the increased importance of their computational requirements, which immediately impacts on their energy consumption. Energy consumption has always been a parameter of extreme criticality in the design of all the embedded systems which depend on a limited source of energy.

First digital embedded system employed no or little software (written in assembly language), their consumed power depending on the different possible operating conditions was relatively negligible. Nowadays, the energy demands of an innovative application is increased a lot with respect of than it used to be years ago. Embedded system comprise large software components, written in high level languages, including a complete operating system with many different network stack and a middleware layer. The load imposed on the system by modern applications such as the decoding of natural or synthetic video sequences, is extremely dependent on the data, and largely variable over different conditions.

In the above scenario, the energy-efficiency of the software components is dramatically more important than in the past. The degree of energy optimization of the software running on a battery-powered embedded device can determine its commercial success, and in some cases indeed its feasibility. It is out of question that even a feature-rich portable product would be scarcely appealing to the customers, if its battery lifetime is short enough to makes it unusable.

Adequate SW support is required to properly exploit the complexity of new systems Nowadays one has to have really good eyesight to understand the trend analysis described so far. In the last couple of years the big "multicore" and MPSoC question has been - how we program these devices? What programming models, tools and methods will exist to let us cope with 1400 processors? Will only the un-embarrassingly parallel applications be able to take use of this SoC complexity? Or can we find ways to make use of all this concurrent processing resource?

Rather than worrying about how to program a device with so many cores using today's thinking, we must be propositional and ask the questions - what kinds of new applications might be enabled with this kind of computing resource? Are there computational models impossible to implement effectively today that this kind of resource might enable?

All these open questions require to deepen invest on software. Has it always happens, hardware support anticipate some possibilities and than software designers are required to improve the system's software counterpart. This time the challenge is particularly tricky because of the different requirement that I have highlighted in the previous section.

Upcoming architectures need specifically designed multi-threaded software to exploit all the potentialities of their hardware parallelism. Multi-threaded software in turn will generally increase the overall applications complexity and relaying on abstraction layers seem to be the straightway to keep that complexity under control.

This two aspects: software parallelization and layered abstraction are two of the most hot topic on actual system's software research.

Software engineers are not parallel thinkers Some researchers have created entirely new software languages that implicitly incorporate parallel programming structures [9]. But in this case the industry has proved itself highly resistant to the adoption of new programming languages. It appears to be very hard to train software programmers to think in terms of parallel, threaded operations.

More appealing appears to be the approaches that investigate on the automatic transformation of single sequential applications into multi thread at run-time [10]. These solutions allow not only to fit well to the large base of legacy software, but can also better support different evolving platforms thanks to their on-line tuning.

The recourse to a virtual execution environment, running on a multi-core processor, which is able to run complex, high-level applications and to exploit as much as possible the underlying parallel hardware, is going to be investigated on some recent works [11] and appears to be a promising approach to effectively tackle the problem of software parallelization.

Operating System should better support user-space To the Operating System (OS) is reserved a special role in the new architectures. Since the early days, the OS has always been the fundamental piece of abstraction from hardware. Being in the midway between more and more complex user-space applications and the upcoming multi- and many-core underlaying hardware architectures it can play an interesting role on better supporting layered abstraction.

In the recent evolution of OS like Linux we noticed the introduction of quite advanced and sophisticated sub-system's specific controls based on local optimization policies. Tools such as CPUFreq [12], CPUIdle [13], the Clock framework [14] and the Voltage Control [15] frameworks are some of the most recent examples of fine detail but almost local controls available in recent kernels. These frameworks are able to enforce quite efficiently a precise control on specific platform subsystems such as CPUs or clock signals. Since the missing of a system-wide view of system resources and requirements, it could happens that these control take local decisions that produce side-effects on some other system components. Side-effects could compromise the quality of control either enforcing only sub-optimal configurations or, even worse, potentially introducing instability on the feed-back control loop.

The burden of bridging the gap, between the low level optimizations and the system complexity, and making possible system-wide optimizations, is entirely on the platform system developer's shoulders. In this regard, system-wide controls have been shown to produce the highest energy gains while hierarchical control systems are considered to be more safe and scalable.

1.2 Requirement for Power and Performances optimization techniques

In the previous section, I clarified and motivated the needs for this research starting from the market and technology's trends. This section is devoted to deriving the requirements starting from designers' needs.

Modern embedded system, especially those targeting multimedia mobile applications are base on more and more complex hardware platforms, providing many functionalities to support multiple and different usages of the same device. When designing and engineering these products, designers are required to provide support for system-wide optimizations that allow to get always good performances and energy consumptions trade-off considering all the competing running applications.

Supporting system-wide optimizations is crucial for many purposes, most notably: to allow multiple functionalities with different competing requirements to peacefully coexist on the same device, and to simplify the development of both applications and device drivers, by concentrating mainly on their functionalities. One of the crucial component interested on providing support for system-wide optimizations is for sure the Operating System (OS). This thesis deals with this very problem:

> *Define a possible system-wide power vs performances optimization strategy, to be implemented at Operating System level, that is sufficiently portable among different platforms without compromising too much its accuracy and efficiency, in all possible different device usage scenarios.*

There are two reasons why I focus my attention to the Operating System level rather than trying to achieve the same optimizations at a different abstraction level: the first reason is that more abstract approaches usually have a lower efficiency and portability; the second reason is that for some specific subsystems the state of the art already provides support for effective local optimizations. I will motivate this claim in the following chapters.

As far as system-wide management is concerned, a platform–independent and general support is still missing and thus a new generation of high-level optimization tools is needed. Applications are becoming larger, more complex and dynamics, and then available optimization solutions are not keeping the pace. I will provide details for all of these claims in the next sections.

With respect to the problem of having a system-wide power vs performances optimization management, new designs need solutions which satisfy the following requirements:

1. system-wide;
2. fine detail;
3. dynamic;

4. scalable;

5. low-overhead;

The following paragraphs clarify what I mean by each of the above requirements. Chapter 2 will show that none of the current approaches satisfies all the above requirements at the same time. Throughout all this thesis, the choices I will make while designing my methodology will be guided and constrained by the above requirement.

System-wide optimization: modern embedded systems, especially those designed for multimedia mobile applications, are based on complex architectures, with many hardware subsystems, and they provide multiple functionalities competing for shared resources. Techniques that only locally optimize each subsystem can be of little practical effect. To get overall system benefits, instead a system-wide approach is required, providing a suitable level of abstraction that don't limits portability while still allowing to keep under control all the specific platforms available resources.

Fine detail: usually different platforms have different subsystems with different available resources and capabilities. Tools that don't fit well with the underlaying system cannot be able to properly exploits its behaviors. Thus a rougher platform detail is insufficient to achieve good control performances because of the risk to miss some optimization opportunities. Tools must be able to exploit detailed description of platform capabilities while still avoiding to become platform-specific.

Dynamic: multimedia mobile devices are becoming more and more dynamic in nature. Different functionalities are provided by the same device which change its role quite frequently depending on the user needs. Moreover we can notice also increasing variability in the behavior of algorithms; the behavior of multimedia encoders and decoders depend more and more on the contents of the streams they process, and applications from many other domains like wireless and gaming show the same trends. Accordingly a good optimization tool must be able to smoothly adapt to frequently changing working scenarios (use-cases) and must be able to keep the actual input data into account.

Scalable: the complexity of devices is increasing continuously, many-core architectures are on the horizon and thus in the future the number of either symmetric and asymmetric processing engine within a single die is set to increase. New generation tools must be designed to be scalable to provide control solutions that can be easily adapted to increasing complex systems without compromising performances.

Low-overhead: optimization techniques should be sufficiently lightweight on monitoring the system and enforcing control decisions on it. Tools that impact too much on system behaviors risk to introduce high latency between system state observation and control actions. This could invalidate optimization actions or even worse it could cost more than what it try to optimize. Low-overhead tools are required, even at the expense of inferior accuracy.

All of the above constraints must be met while keeping reasonable good performances for the control problem of trade-off power vs performances. Relative control accuracy is important especially when designers need to compare alternative solutions. Absolute accuracy is important especially when designers want to evaluate the performance of a control solution on different architectures.

Unfortunately, as the sections dedicated to the related work will show, none of the approaches currently available tackle appropriately all the above requirements at the same time. For example, drivers based approaches does not fulfill the dynamism and system-wide requirements. Current user-space approaches do not satisfy the detail requirement. An informal summary of the above considerations is given by the table below:

Abstraction	System wide	Fine detail	Dynamic	Scalable	Low overhead
drivers	✗	✓	✗	✓	✓
centralized	✓	✓	✗	✗	✓
application	✓	✗	✓	✓	✗

Table 1.1: How existing techniques fit the requirements for a good power and performances optimization tool

All these reasons motivate the crucial need for system-wide, fine-detailed, fast, dynamic and accurate operating system support for power and performances trade-off optimizations.

This thesis is dedicated to the research of a system-wide power vs performance optimization control which fulfill the above requirements.

1.3 Why we focus on Linux

The choice of an Operating System in which implement our proposal for a new system-wide optimization techniques is a crucial element for this work. I don't want just to provide yet-another-theoretical algorithm but instead I want to show how the proposed technique can be effectively implemented on a real system.

The technique I propose is largely independent from the Operating System which it is applied. Although I show an instance of this technique which is implemented as a Linux kernel framework, the technique does not rely on any specific feature which is provided by Linux only. Porting the same technique to another operating system, especially those written in C languages, in most cases is just a matter of few interface changes.

Nevertheless, the choice of the Linux kernel for the particular instance of the technique which I show in this thesis demands some justification. I choose Linux because it is OpenSource and because Linux, in all of its flavors, is one of the

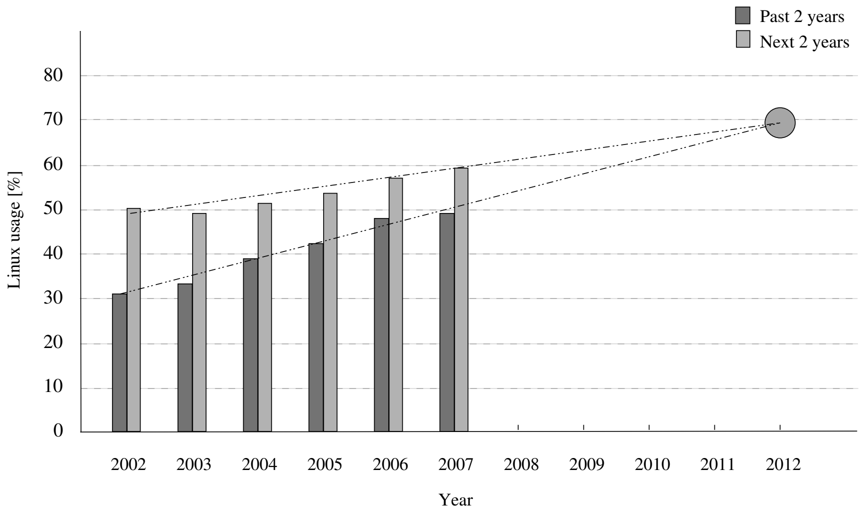


Figure 1.3: Actual and planned Linux use in the development of embedded systems may converge by 2012. Source: LinuxDevices.com.

leading operating systems in the embedded design community, and more and more embedded systems feature some Linux distributions.

Figure 1.3 present an interesting outcome from a study that covers the worldwide market for Linux employed in the development of embedded systems. This study has been conducted by the authoritative LinuxDevices.com website, dating back 2007 and compare the planned Linux use over the next two years, as well as actual use over the previous two. The study comments on the increasing use of Linux and forecast that this predominance is not to arrest soon, and motivate why:

> «[. . .]in the early days, planned use far surpassed actual use. Today, as more and more project teams succeed in executing their Linux migration strategies, the Linux "uptake gap" has narrowed dramatically. Trend lines on the chart below suggest that by 2012, actual and planned Linux use will converge, at about 70 percent[. . .]
>
> Interestingly, last year's survey data suggested that embedded Linux's uptake gap would close between 2009 and 2010 at closer to 60 percent. However, some 61 percent of this year's respondents plan to use Linux within two years, compared to 58 percent in the past two years – an increase that suggests Linux adoption may not level off as quickly as previously believed, and may achieve a greater overall market penetration than originally thought.»

The above figures and comments suggest that the diffusion of Linux kernel usage on embedded systems is still long, and that a system-wide power and performances optimization technique based on this Operating System, like the one proposed here, will be of great practical usefulness still for a long time to come.

1.4 Many techniques are possible, just one is chosen

First, from now on I will generally use the term *framework* to denote an operating system's component that provide system-wide support for both power and performances optimizations. Carefully reducing overall system's power consumptions require a system-wide framework. Running concurrently resource-competing user-space programs while ensuring expected and perceived Quality-of-Services can take advantages from a system-wide framework support. All this thesis is about the definition of a system-wide framework, in this sense.

The goals for a system-wide power and performances control framework can be carried out in a variety of ways. The very purpose of such a framework is to "build a model" of the target system that can be used to identify the optimal system configuration according to run-time informations on resources availability and application's requirements. This is generally a complex process, which may involves also modeling how the underlying architecture works on different usage scenarios (use-cases).

The properties of a model basically depends on the abstraction level we consider. More detailed models are more complex (therefore more difficult to build and to maintain, and slower to execute), but they may ensure better accuracy. All the possible modeling choices can be represented and compared in a complexity/accuracy space. Among these models, the ones which are non-Pareto-optimal do not deserve any further attention, because they are surpassed by some other model in both the objectives. The remaining ones generate a discrete (very large) set of models which I intuitively represent in Figure 1.4.

The Pareto-optimal solutions have been classified according to some evaluation indexes. I chose to focus on some index which are someway related to requirements identified in the previous section. The 'simplicity' index is an evaluation of how simple it is to "use" the framework both from the user-space perspective and the platform integration. The 'control precision' is quite self-explaining, it measure how good is the framework on approximating the real optimization goal avoiding local sub-optimal ones. The 'run time efficiency' is directly related to the 'low overhead' requirement, a framework is as much run-time efficient as much it don't affect on system performances which is trying to optimize. The 'platform independence' and 'scalability' indexes describe how well the framework can easily adapt to different platforms with almost minor modifications required and how it can easily scale with increasing system complexity. Finally the 'simplicity' index is an abstract evaluation of the framework implementation complexity. This last index has been explicitly considered because it is undoubted that conceptual simpler framework

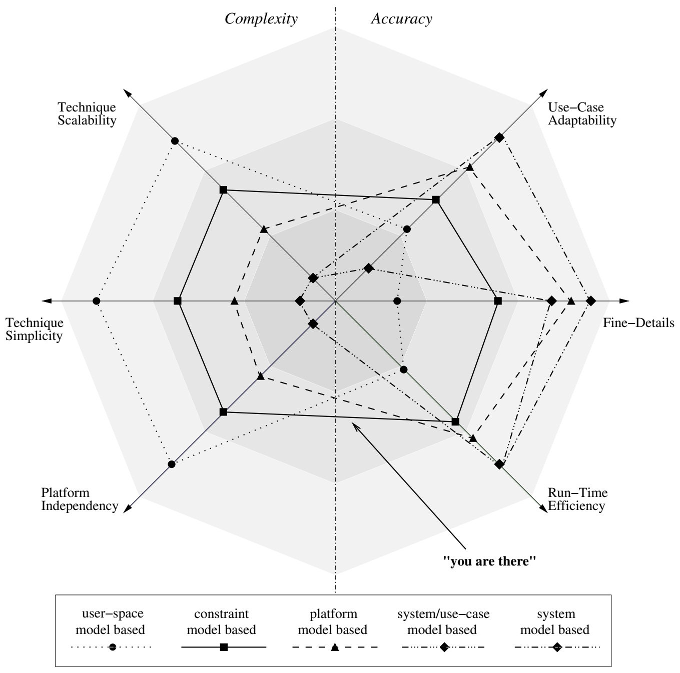


Figure 1.4: Even after discarding the non-Pareto-optimal ones, many techniques to develop a system-wide optimization framework are possible. They exhibit different trade-offs between simplicity and accuracy. In this thesis I choose one, which is particularly suitable for multimedia mobile embedded systems.

are more probable to be endorsed by community and thus have higher probabilities to be well accepted and integrated within existing systems, e.g. the Linux kernel.

In the Figure 1.4 are compared with regard to these indexes, the main Pareto-optimal techniques which can be identified in bibliography. Techniques based on a 'system model' (dash-dotted curve with rhomboidal points) are those that grant an overall maximum accuracy. These techniques usually relay on an off-line built model that capture the system behaviors and that can be used at run-time as a black-box controller where application requirements are the inputs and the optimal system-wide configuration is the output. Accuracy can be further improved, as indicated by a better 'use-case adaptability' index, when using a variant of these techniques which is based on 'use-case model'. In this case a system model is built for each use-case and selected at run-time by the control policy according to the specific working context. The main downside of these techniques is the complexity: building a system-wide model usually require extensive simulations in order to collect enough profiling data for post-process with some mathematical or statistical tool. The validation of a model also require an extensive work. Moreover, the models obtained are typically highly platform dependent and substantially unmodifiable without redo all the simulations. The end complexity is so high that usually these techniques are hardly adopted for multimedia mobile devices and are instead used on mission-critical embedded systems with longer life-time.

A significant complexity mitigation without compromising too much accuracy can be obtained using techniques based on a ’platform model’ (dashed curve with triangular points). These techniques in the past have been effectively developed also for multimedia mobile devices, although they have not had a big spread. In this case the model is build as part of the platform-code and thus, being a software defined model, it can be modified and adapted to different platforms more easily. The main limitation is still the complexity because building the model require a detailed knowledge of all hardware subsystems.

The simpler techniques are those based on a ’user-space model’ (dotted curve with round points). In this case the control policy is usually highly abstract from the underlying hardware and thus they require to trade accuracy for a reduced complexity. These techniques are easily portable among different platforms, usually even operating system independent and running as simple user-space applications. On the other hand, unfortunately these techniques usually show a too small accuracy to effectively support the aggressive optimization required by modern multimedia mobile devices.

A quite good trade-off between complexity and accuracy can be obtained using techniques based on a ’constraint model’ (continuous curve with squared points), usually based on a distributed control policy that allows to reduce the complexity. Indeed, utilizing the principle of "divide and conquer", the complexity could be distributed on multiple modules such that each single one turns to be more simpler to define. This approach imply a modular design which improve scalability and portability too. The accuracy of that models with respect to the system-model based ones is reduced but still acceptable for multimedia mobile devices. This is mainly due thanks to the relatively high fine-details level that are still possible to reach with such kind of solutions.

The space and time allowed by a doctoral thesis permits the complete analysis of just one of these many techniques. I choose to focus my attention on the “constraint model” approach. This choice has to main motivation: the lakes of an established and well defined framework of this type for the mobile multimedia embedded devices application context, and secondly because it show the most promising application possibilities in this specific application fields. In instance, the fundamental approach presented here can be extended and refined to support some levels of HW accelerations. Anyway these possibilities are just going to be investigated deeply and I will not support these claim in this documents, as it would require too much time and space.

1.5 The fundamental approach of this thesis

In this section I provide a quick overview of my approach. I summarize its fundamental steps and motivate their choice in terms of the requirements expressed in Sec. 1.2 on page 12. The same reasoning is also summarized in Fig. 1.5.

The 'system-wide' and 'fine detail' requirements motivate the decision to have an in-kernel framework. A framework at this level, which sit in between user-space applications and the bare hardware, can easily exploit the overall view on the system state while still not loosing too much details of the actual platform. Moreover kernel space already offers a set of highly efficient and optimized support functionalities that can be used for the implementation of a sufficiently lightweight optimization framework.

The low level details about the specific platform resources and capabilities are provided by both platform code and drivers, which are directly accessible by an in-kernel framework. The platform code, i.e. the architecture specific code that a kernel like Linux require to implement in order to have a very basic level of abstraction from the underlying hardware, can define a set of platform-specific metrics (PSM). These metrics represent platform configuration parameters to tune system behaviors and that are usually directly controlled by some drivers. Drivers in our view play a central role because they provide both some data to the optimization framework, where are used to reconstruct the specific system-wide view on capabilities and resources, and also implement highly efficient local optimization policies, which allow to have a fine control on the specific hardware.

The in-kernel framework is the perfect point to make a "divide and conquer" operation. Although it is not generally true that the coordination of local optimization policies leads to the same performances of a single centralized optimization control, this assumption leads to negligible worse while to the contrary show interesting benefits for overhead reduction and scalability of the solution. This motivate the decision to opt for a hierarchical control architecture, which has two kinds of control policies at different abstraction levels. Each driver implement its own low-level control policy, which is 'fine detailed' and 'low overhead'. A more abstract control policy is implemented by the in-kernel framework, which in turn is 'system-wide' and grants the 'scalable' requirement.

The framework's system-wide optimization policy satisfy the 'dynamic' requirement. This policy allows to implement a coarse-grained control that identify the high-level system-wide optimization strategy; which in instance could be either a performance boost or power reduction or even a mix between these two goals. Policies that are local to each driver provide fine tuning, within some operating mode enforced by the global policy, with very low run-time overhead and exploiting the 'fine details' knowledge about the controlled hardware. Moreover, the system-wide policy allows to dynamically adapt to changing working conditions by simply changing accordingly this policy. Different optimization goals can thus be enabled at tun-time.

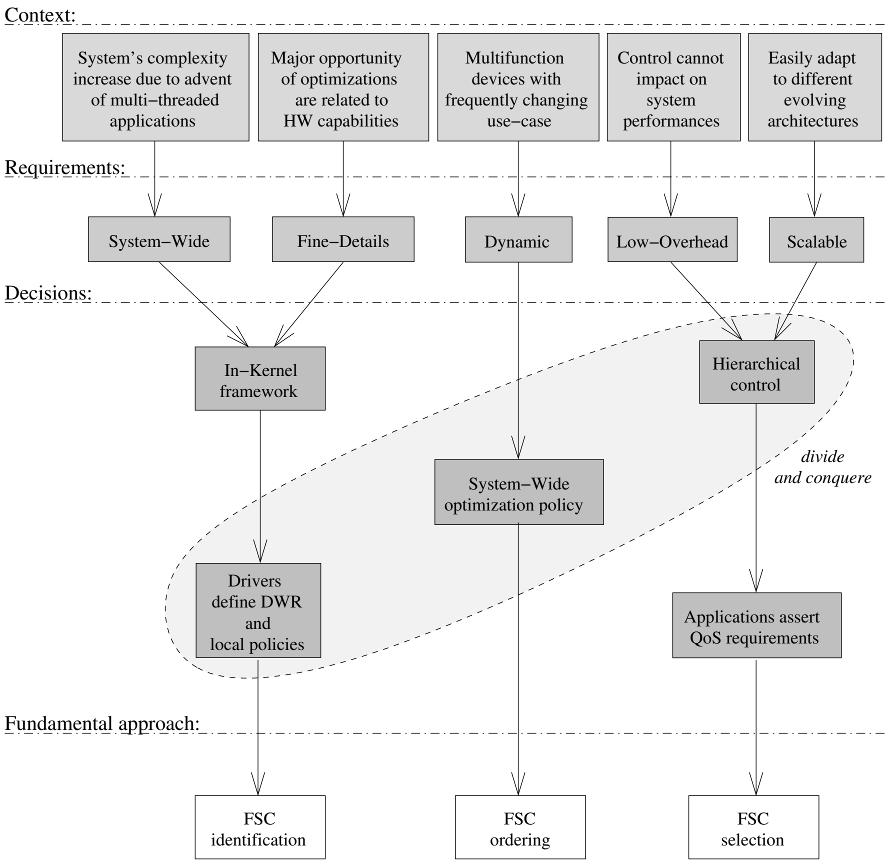

Context:
System's complexity increase due to advent of multi−threaded applications
Major opportunity of optimizations are related to HW capabilities
Multifunction devices with frequently changing use−case
Control cannot impact on system performances
Easily adapt to different evolving architectures
Requirements:
System−Wide
Fine−Details
Dynamic
Low−Overhead
Scalable
Decisions:
In−Kernel framework
Hierarchical control
divide and conquere
System−Wide optimization policy
Drivers define DWR and local policies
Applications assert QoS requirements
Fundamental approach:
FSC identification
FSC ordering
FSC selection

Figure 1.5: How the fundamental approach of this thesis derives from the designers' requirements which, in turn, derive from the current context in the embedded design scenario.

From all these considerations and consequently decisions I have derived the fundamental approach of the technique developed in this thesis. This fundamental approach: accept inputs, yield outputs, and is composed as I describe below.

The technique shall get as inputs:

- a platform description, in terms of platform-specific system-wide metrics
- the driver capabilities, which define how each device mode maps on system-wide metrics
- an optimization policy, which can be changed at run-time and define the optimization goal
- QoS run-time requirements, either from user-space applications or platform subsystem, which define some constraint for the optimization problem

It shall yield as output an estimation of the optimal Feasible System-wide Configuration (FSC), taking into considerations resources availability and applications requirements.

It shall be composed of three fundamental phases:

- FSC identification
- FSC ordering
- FSC selection

More detailed explanations of the above steps follow.

In the ’FSC identification’ phase, the platform specific information collected both from platform-code and drivers are processed in order to automatically identify all the possible system-wide configurations which are feasible (FSC) for the specific platform. A FSC is defined by constraining a set of system-wide metrics while granting that all devices have an operating mode compatible with those constraints. This step is, in fact, one of the more complex topics of this work. I will show how this problem could be tackled and solved with different levels of efficiency.

In the ’FSC ordering’ phase, a weight is associated to each identified FSC according to the running coarse-grained optimization policy. This weight is a measure of the optimality of an FSC with respect to that policy. Since policies can be changed dynamically to better adapt to different working conditions, in turn the weights also can change at run-time with respect to the new optimization goal.

In the ’FSC selection’ phase, among all valid FSC just one is chosen, according to actual resource availability and optimization goal, and then enforced as the optimal system-wide configuration. At run-time, not all FSC identified during the first steps could be valid at any specific time. Applications or subsystem requirements could invalidate some of them. During this step the framework should be able to efficiently identify still valid FSC and select the optimal one.

1.6 The final objective of this thesis

In the light of all the above considerations,

> this thesis researches a system-wide, fine-detailed, dynamic use-case adaptive, scalable and low-overhead framework to support the identification of an optimal trade-off between expected performances and reduce power consumptions on mobile multimedia embedded systems running a general purpose operating system.

The framework must be implemented on updated Linux kernel, exploiting already existing common facilities, and provide a simple interface that facilitate integration within existing device drivers. The define user-space interface should be platform-independent in order to support portability and minimize needs for modifications by developers. The framework should be able to suggest which are the optimal system-wide configurations according to actual available resources and competing use-space requirements. Moreover, as a further extension, the core framework should support design and configuration exploration which allow the inspection of eventually sub-optimal working modes and drive either software or platform redesign choices.

1.7 A bird-eye-view on the proposed technique

In this section I provide an high-level view of the proposed approach.

The distributed control approach allow assumptions which facilitate modeling, more precisely: local policies are able to identify optimal device configurations give a set of constraints which usually change only at use-case changes. These assumptions are quite reasonable, especially when we look at multimedia mobile applications. In this specific case it's common to have highly specialized device drivers running fine tuning optimization policies and the typical usage of the device change during time but with a slower dynamics when compared to that of each single device.

These assumptions justify the recourse to a distributed control model, which is based on different levels of abstraction. Moreover, a distributed approach allows very good run-time accuracy, both in term of precision and overhead, while still not increasing to much the framework complexity, and thus avoiding to reduce scalability or portability.

In Figure 1.6 is showed how different elements of the technique I propose correspond to different abstraction levels. The main elements are: *control policies*, operating system's *elements* and the *entities* introduced by the proposed framework. The lower level component is represented by the platform code, which could define some platform specific control parameter (i.e. Platform System-wide Metric, PSM) in addition to platform independent ones (i.e. Abstract System-wide Metric, ASM)

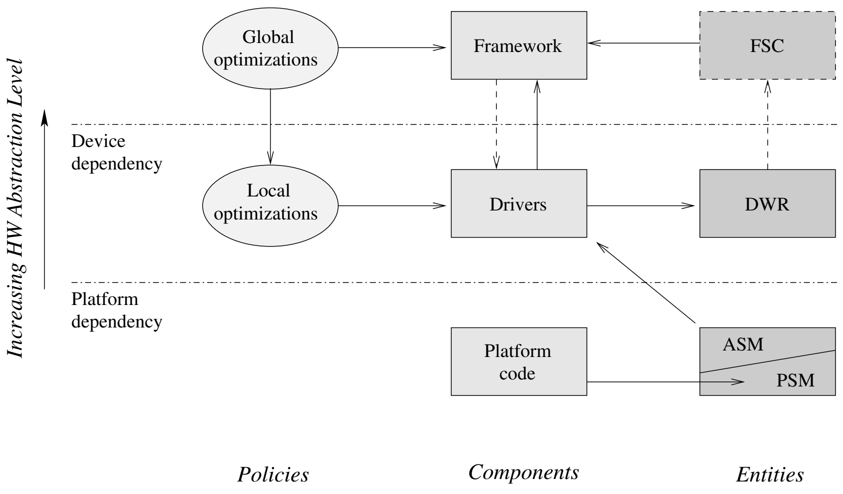


Figure 1.6: The proposed technique is developed at different abstraction levels. Lower levels exploit HW details to allow fine tuning while higher levels ensure system-wide optimization by exploiting run-time informations to derive coarse grained requirements for the lover levels. The different abstraction levels can be identified both on policies, operating system's components and framework defined entities.

defined by the proposed framework. At an higher level of abstraction we find local optimization policies, which are strictly related to device drivers. Each device driver could have its own fine tuning optimization policy and refer to System-wide Metrics (ASMs and PSMs) to define its operating modes (i.e. Device Working Regions, DWR). The maximum abstraction level is completely platform and devices independent and define global optimization policies. These policies, whit respect of local ones, are system-wide and thus will allow to exploit a complete system state view in order to identify the better trade-off between power consumptions and system performances. The framework code is in charge to run global optimization policies and enforce decisions to underlying drivers which could use them as some coarse grained optimization requirements. To properly run global policies the framework exploit the knowledge of all of the system feasible configurations (i.e. Feasible System-wide Configuration, FSC), which can be automatically identified starting from some informations about drivers and theirs working modes.

An overall view of the architecture for the proposed approach is showed in Figure 1.7, where: policies, operating system's components and entities are localized within the different software abstraction levels of a generic operating system. The framework along with the platform code define the set of available system-wide metrics (ASM/PSM), which in turn drivers refer to define theirs operating modes

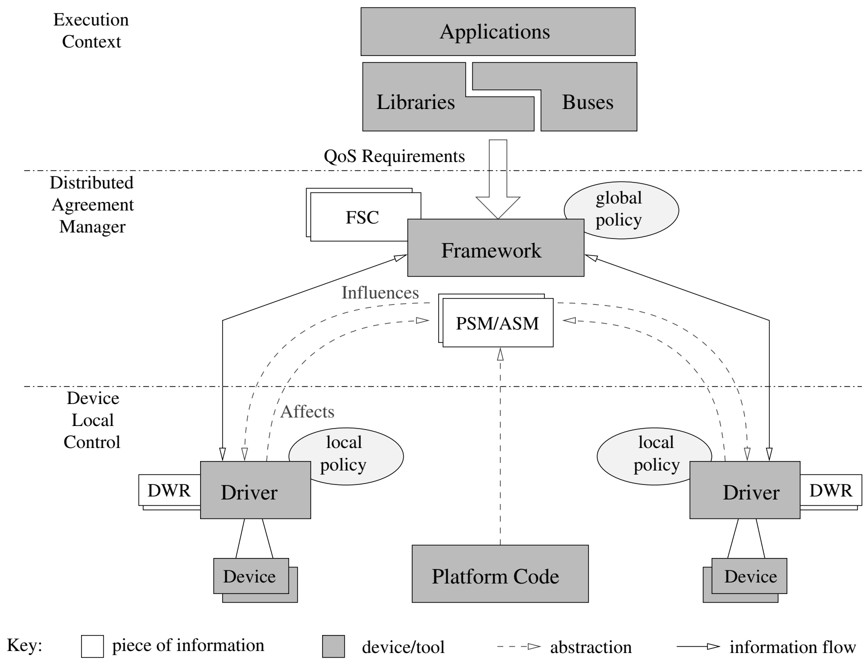


Figure 1.7: A bird-eye-view of the proposed system-wide control architecture. The main system components are represented by platform code, drivers and the optimization framework code itself. These components either define or use some entities: system-wide metrics (ASM/PSM), device operating modes (DWR) and feasible system configurations (FSC). Policies are both locally and globally defined and are implemented as modular components so that can be changed at runtime.

(DWRs). Local policies can be associated directly to each driver while a global optimization policy is provided by the framework itself. Such an architecture provide a single and well defined interface to the userspace that can be used to collect Quality-of-Services (QoS) requests directly from a generic "execution context". The execution context is represented by applications, usually defining different use-cases, or better it can be represented by software libraries and buses. The eventually instrumentation of these last software components is particularly interesting because it allows to extend transparently the benefits of using such an optimization framework to all application using them.

1.8 Advantages of this approach

The approach proposed here offers a number of advantages with respect to its competitors:

- distributed control:
  the framework I propose support a system-wide optimization based on a distributed control model. This approach allows to split frequently device specific fine tuning controls from less frequent coarse grained controls related to use-case changes. Single devices' fine tuning is demanded to drivers local policies which can exploit fine details knowledge about the underlying hardware. These local policies are constrained by system-wide requirements identified by a global optimization policy running at an higher abstraction level. The global optimization policy provide a coarse grained but system-wide control which ensure to identify global optimum configuration while still not impacting to much on overall system performances.

  Moreover, distributed control allows also to allot control complexity on different components, basically on each device drivers, and support their local policies with just some constraints coming from a system state overall view. The distribution of the complexity also allows you to implement solutions that are modular and support composition and reuse of code. This at the end improve the simplicity and the portability of the final solution.

- improved portability:
  the modular implementation of a distributed control model improve the portability to different platforms and systems, of an existing solution adopting the framework I propose. Indeed this requires to change only low-level components, usually drivers and their policies, to match the target system, while its possible to reuse the code for common components. Since the control model don't use a predefined system-wide model, but instead is able to automatically build such a model at run-time, the modeling effort is also reduced and substantially limited to drivers' local policies. This effort is worth done because at this level of abstraction not only it is possible to exploit fine-details, producing highly device-specific and optimized policies, but also the produced code will be reusable on all platforms using the same device. This is possible thanks to a properly defined interface between the proposed framework and drivers.

  A domain expert could claim that in practice it is very difficult to extract the FSCs for all the components and the inter-dependencies between them, also considering that they all come from different vendors. Regarding this point it is worth to notice that, one of the main advantages of the proposed framework compared to others is rights its ability to compute automatically all these information. Moreover, this is done starting from much more simpler information which require only a restricted knowledge on single subsystems or device drivers.

- simple integration:
  while the drivers' interface support code reuse on different platforms, a properly defined interface that the framework I propose expose to the user-space allows to increase even more platform independence for this side. The use of well defined interfaces to decouple software components simplify the integration of both drivers and user-space with the framework. Each software component will be required to implement only few and specific extensions in order to correctly take part to the optimization control.

- different level of abstraction:
  the technique proposed exploits different levels of abstraction in order to implement a "divide and conquer" approach. The low levels of abstraction refer to specific components and can utilize their knowledge in order to finely adjust devices' operating modes. The more abstract components can instead use a comprehensive system view to ensure the achievement of a global optimization. The "different levels of abstraction" approach has been extensively adopted in the proposed techniques for the definitions of all its elements. Policies have been spitted in locals and global. Operating system's components range from low level platform code trough drivers up to the framework code and user-space applications. And finally, entities defined by the proposed techniques corresponds to different levels of abstractions with platform specific metrics (PSM) defined by the platform code, abstract metrics (ASM) and devices operating modes (DWR) defined by the drivers and finally the feasible system configurations (FSC) which are automatically identified by the framework.

  The layered design of the proposed approach allows also to improve its run-time efficiency. Indeed, the framework require to run algorithms of different complexity ad different timeframes. The more complex operation is FSC identifications, but it is run just on time after system boot. An average complexity function is the FSC ordering, but it happens only when the optimization policy changes and this is a quite infrequent event. Moreover, this operation allows to speedup the most critical operation: the FSC selection. This is also the most efficient algorithm, with a very limited run-time overhead. Experimental results show that we could aim to handle a real system with a million of FSC with a negligible impact on performances.

- maintainability:
  all the previous points contribute to simplify the maintainability of an implementation. Either changing/upgrading a single system component or considering a new use-case does not require extensive software modifications but only adjustments limited to the specific driver or corresponding software

component. The framework then provides all the mechanisms to correctly reconstruct an updated system-wide state view and thus ensure the global optimization policy to run correctly.

- different working mode:
  the proposed technique support different working modes to better fit target platform capabilities and progressive integration. A *best-effort mode* allows rapid prototyping when porting the framework on a platform with not all drivers integrated. In this mode the framework is still able to feed requirements to already integrated drivers supporting their local policy but not granting global optimization. A *distributed-agreement mode* instead can be enabled when all interested drivers are integrated with the framework and is able to identify global optimum and force drivers to always agree on a new feasible configuration once the use-case or system requirements changes.

- integration with operating system:
  an implementation of the proposed framework has been developed and completely integrated within the Linux kernel. This will ensure a maintained code-base implementing the main mass of the whole framework, constantly revised and improved by the community. Moreover, new drivers policies, once developed end integrated are always available and ready-to-use for new platform's development thanks to the modular approach provided by the framework design.

1.9 Frequently raised objections

When working in a research area which is populated and commonly approached with a quite different mentality with respect to the relatively new approach you are advancing, it is often difficult to convince one's own audience about the novelty of his approach and its soundness. Power and performances trade-off for embedded system is for sure such a field. Additionally, in the specific topic covered by this thesis, research papers are relatively few and often "overclaim" their accomplishments in their titles, with respect to the actual contribution described in their texts. This practice pollutes the research concept space, inducing the idea that a problem was effectively solved when it was in fact just described, or tackled incompletely. Most of the time the complex problems we are facing is dissected and the research focus is only on specific subsystem, without considering the overall effects, or even worst basing the solution on a set of hypothesis which ten turns to be too much constraining to have a realistic and usable solution.

This section is designed to help reviewers not being mislead by these practices, and to anticipate criticism. It is therefore structured as a "Frequently Asked Questions" section of a manual. It is written in an informal, straight-to-the-point style. The reader will indulge me, as long as my claims are correct and motivated.

1.9.1 «Your novel contribution is not quite clear»

This work has novelty in its objective, in its method and in its results. The scientific merit is obviously in the method. This technique is the first work which attempts the definition of a system-wide framework, for distributed control of the trade-off between power consumption and performances, actually implemented and verified, which accomplish that objective. The framework comprehensively considers all system's elements, at any abstraction level, without compromising neither performances nor precision thanks to a proper modular design.

1.9.2 «CPM has already been implemented!»

Yes and no. The matter is ambiguous. It all depends on how you define "constrained power management". Since this ambiguity is a potential danger to understanding the contribution of this thesis, then I prefer to disambiguate the term.

I motivated above why new platform design need a technique to support the optimization of power consumption vs performances trade-off, and such technique must relay on a system-wide approach (i.e. it must identify global optimum). Because of this need, I define "constrained power management" the ability to identify system-wide optimal configurations, considering both different low-level capabilities and changing high-level resource requirements at the same time.

According with this definition, this thesis present **the first** constrained power management technique. No other techniques before have provided fine-grained control without resorting to lower level platform models (typically built by off-line profiling and usually considering only a limited number of sub-systems). On the other hand, all the techniques which addressed system-wide optimizations, were unable to consider neither fine-details for the configuration of devices or dynamic use-cases.

The terminology is debatable, you may apply broader definitions of the "constrained power management" term. In this case, a number of works in literature may be considered to have already applied this technique. Nevertheless, the above considerations must be kept clearly in mind, otherwise the novel contribution of this work is neglected just for lack of precision in terminology.

1.9.3 «Your approach is too limited»

This thesis presents a broad methodology which define many components and some platform specific extensions, but it details only the core framework, the global optimization strategy and the interface for the extension. This instance is incomplete to be effectively used on a real-system but it completely define the overall common components so that the definition of extensions is simple and precisely defined. I did that because I want to stick to a good principle: do one thing, do it well.

I claim that a solid but narrow theoretical foundation which may be extended with some effort is more desirable than a set of broader but less solid foundations. It is clear that in the scope and duration of a single Ph.D. thesis it is impossible to achieve breadth and solidity at the same time. That is obviously the ideal goal, but not practically achievable. To some extent, generality can be traded with accuracy, and I have chosen to privilege accuracy.

1.9.4 «Your approach is too difficult to actually use it»

I hardly tried to keep simple the proposed techniques, indeed the integration simplicity is reasonably considered a key factor for the success of any new proposal. Nevertheless too much simple solutions hardly can grant good performances especially regarding control accuracy. All that considered one of my design constraints has been to better support a "divide and conquer" approach in order to spread the complexity too.

The main complexity of the proposed approach has been located within the framework core, where the global optimization policies and proper interfaces toward both user-space and drivers are implemented. What still remains on the platform designer's shoulders is just the platform specific code and the low-level device drivers' policies. It is worth to notice that those components are completely independent each other thanks to the modular design of the proposed approach. This allows to assign specific extensions to different and well prepared subsystem experts: each one could share his better knowledge of a specific device working modes with others by simply providing a self-contained integration to the framework. The framework itself will then provide all the necessary support to collect all those fine-detailed informations and exploit them the better is possible, compatibly to run-time available resources and use-case requirements, to ensure system-wide optimization in an almost completely transparent way.

1.10 The organization of this thesis

This thesis is organized as follows.

In this chapter, I have proposed a fundamental approach to the power and performances trade-off control for multimedia mobile systems. The fundamental approach is broad and may lead to a large number of specialized instances, targeting narrower domains and internally relying on different sub-approaches.

In Chapter 2 I present, for the research areas related with the main objectives of this thesis, the most important works present in the literature. For the works which present competing approaches, I try to give a critical comparison, illustrating advantages and shortcomings.

In Chapter 3 I present the complete details of one of the possible instances of techniques inspired by my fundamental approach. The Chapter discusses the ideas which are at the basis of proposed technique, and all the steps which allow to realize it. A possible implementation for the proposed approach is presented in Appendix A. Here I give also an implementation of the extension point of the framework to show how it can be effectively used by a real embedded platform.

Chapter 4 presents experimental results which prove the accuracy and utility of the technique presented, it draws final conclusions on the quality and breadth of the theory and results proposed here, and it sketches the current and future developments on the topic.

Chapter 2

Background

"A man who knows how little he knows is well, a man who knows how much he knows is sick."

Witter Bynner

Increasing computer performances has always been a main research topic. Although the main concept of MOSFET transistor is the same since its invention by Frank Wanlass, in 1963 [16] the semiconductor production technology has been evolving constantly, particularly in the direction of miniaturization.

Despite advances in semiconductor technology fueled the tremendous increase in transistor density, power dissipation constraints coupled with limits in the instruction level parallelism (ILP), have caused the high performances computing architectures roadmap to enter the multi-core and many-core era. The era of chip multiprocessing (CMP) started with a step-back in core operating frequencies, using multi-core chips with shallower processor pipelines [17, 18]. This allowed affordable powers while still not constraining to much the multi-core throughput to increase. However many crucial application domain still require single-thread performance increase and the growth in singe-chip's cores number cause a super-linear growth in total core area. This two aspects determined a corresponding power consumptions increase and thus the power consumption problem did not disappeared in the new era of multi-core.

Modern computing architectures are far more powerful and versatile than their predecessors but they also require a lot of power. Power densities have increased rapidly and correspondingly have increased concerns about chip reliability and their life expectancy, cooling costs and even environmental aspects, posing new challenges in the design techniques of chips.

Nowadays, although other constraints like chip I/O bandwidth and inter- and

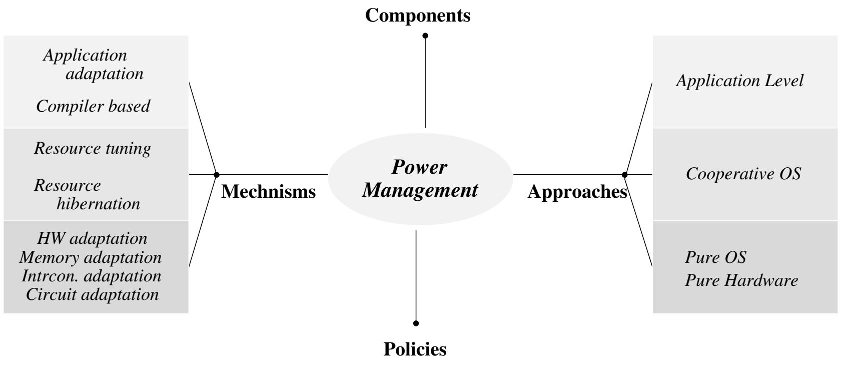


Figure 2.1: An high-level taxonomy of prominent power management mechanisms, approaches and policies. Mechanisms and approaches can be implemented at different abstraction levels: architectural, middleware and software. Cross-Layer techniques are approaches that try to exploit mechanisms from different abstraction levels at the same time. Policies instead allows to implement different instance for the same approach-mechanism pair.

intra-chip data bandwidth begin to emerge as new secondary limits, power and peak temperature still continue to be the key performance issues. These limiters pose even more serious issues for small battery powered mobile devices. The growth in the scale, complexity and flexibility of the applications required on portable, battery-powered embedded systems, is even more impressive. Unfortunately battery technology has not and cannot keep pace with the demands that new generations of mobile devices [19].

Power management is a multidisciplinary topic, that involve many complex aspects (e.g. temperature, reliability, battery duration, ...) at different abstraction level. Techniques to reduce power consumption in computing systems range from physical layers design up to higher software abstraction levels. An high-level taxonomy of prominent techniques is illustrated in Figure 2.1. Cost-effective solution for the power reduction problems require to tackle it at all the abstraction levels simultaneously. Despite that the lowest level of abstraction I treat is right over the physical layer of a system and deals with the design of chips. Thus I don't discuss the physical properties of materials used in the production of chips, which is the matter of advanced and specific research field. Anyway some elements belonging to the circuit layer, namely transistors and logic gates, play an important role on both understanding where power is consumed and how some techniques could be effectively applied to reduce energy wastage.

In the rest of this chapter, some basic concept about power consumption are briefly presented, and then I will review only the layers which are more relevant for my work, while I forward the reader to the available bibliography [20, 21, 22] for a presentation on the lower layers optimization techniques.

2.1 Defining Power Consumption

Power and energy are commonly defined in terms of the work that a system performs. Energy is the total amount of work a system performs over a period of time, while power is the rate at which the system performs that work. By defining P the power, E the energy and W the total work performed in a specific time interval T, we can write:

$$P = W/T\ [Joules] \tag{2.1}$$

$$E = P * T\ [Watts] \tag{2.2}$$

In the context of a computing system: the work is defined by the activities associated with running programs, energy is the total electrical energy required (or dissipates as heat) by the system and power is the rate at which the system consumes electrical energy while performing its activities.

The distinction between power and energy is important because techniques that reduce power do not necessarily reduce energy. For example, the power consumed by a computer can be reduced by halving the clock frequency, but if the computer then takes twice as long to run the same programs, the total energy consumed will be similar. Whether one should reduce power or energy depends on the context. In mobile applications, reducing energy is often more important because it increases the battery lifetime. However, for other systems (e.g. servers), temperature is a larger issue. To keep the temperature within acceptable limits, one would need to reduce instantaneous power regardless of the impact on total energy.

The total power consumed by a computing system can be differentiated into two components: *static power consumption* and *dynamic power consumption*. These two components have different nature [23] and corresponding optimization techniques.

2.1.1 Dynamic Power Consumption

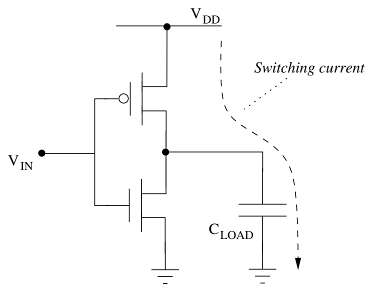
Dynamic power consumption arises from circuit activity such as the changes of inputs in a gate or values in a register. It has two main sources, switching power and short-circuit power.

Switching power is the primary source of dynamic power consumption, which is the power required to charge and discharge the output capacitance of the gate of a transistor as depicted in Figure 2.2a. This power component can be formulated as:

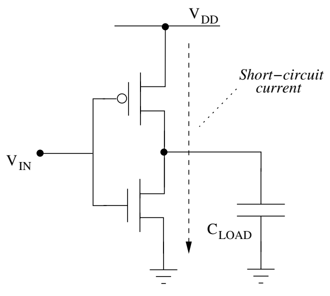
$$P_{dyn-switch} = Energy/Transition \cdot f = C_L \cdot V_{DD}^2 \cdot P_{trans} \cdot f_{clock} \tag{2.3}$$

where the $Energy/Transition$ is defined by the product of:

- C_L - the load capacitance
- V_{DD} - the supply voltage



(a) Switching power



(b) Short-circuit power

Figure 2.2: The main components of dynamic power consumption. *Switching power* (a) is the primary source of dynamic power consumption and arises from the charging and discharging of capacitors at the outputs of circuits. *Short-circuit power* (b) is a secondary source of dynamic power consumption and accounts for only 10-15 consumption.

while the transitions' frequency f is defined by the product of:

- P_{trans} - the probability of an output transition
- f_{clock} - the system clock frequency

Since switching power is not a function of any specific physical parameter, but rather a function of the activity a load capacitance, it is data dependent.

Internal power is a secondary source of dynamic power consumption Fig. 2.2b, and is related to the short-circuit current. This current arises because circuits are composed of transistors having opposite polarity, negative or NMOS and positive or PMOS. When these two types of transistors switch state, there is an instant when they are simultaneously on, creating a short circuit which correspond to a current flow. Internal power is given by the equation:

$$P_{dyn-internal} = t_{sc} \cdot V_{DD} \cdot I_{peak} \cdot f_{clock} \tag{2.4}$$

where the two new components that appears represent:

- t_{sc} - the duration of the short circuit current
- I_{peak} - the total internal switching current

Summing these two main components, we obtain this formulation for the dynamic power consumption:

$$\begin{aligned} P_{dyn} &= P_{dyn-switch} + P_{dyn-internal} \\ &= (C_L V_{DD}^2 P_{trans} f_{clock}) + (t_{sc} V_{DD} I_{peak} f_{clock}) \\ &\sim aCV^2 f \end{aligned} \tag{2.5}$$

As this equation shows, the more dominant component of dynamic power depends essentially on four parameters namely, supply voltage (V_{DD}), clock frequency (f_{clock}), load capacitance (C_L) and an activity factor (a) that relates to how many transitions occur in a chip. Instead, the internal power component can be neglected as long as the duration of the ramp of the signal switching is kept short because the short circuit current occurs only for a short time for each transition, so the dynamic power is dominated by the switching power component of the last equation.

Parameters that can be manipulated in order to reduce dynamic power consumption are voltage, frequency and the data-dependent switching activity. Particularly, observing the expression of P_{dyn}, the reader may note the quadratic dependence of power on the supply voltage, thus, decreasing V_{DD} is a widely used way to reduce dynamic power even if this technique should be use with care because the speed of a gate decreases with the supply voltage. In instance, this technique could be used for circuit blocks that do not have to run particularly fast, like in peripheral devices, introducing the use of different voltages for different areas (subsystems) of a SoC, this approach is also know as multi-voltage. Instead, for fast circuit like host processors and accelerators, designer can provide variable supply voltage with respect to the load on the chip and to the computing requirements in every moment: higher voltage, and related faster clock for high performances tasks and lower supply voltage with lower frequency for less performance demanding tasks, this approach is called voltage-scaling. Another approach for reducing dynamic power is clock gating which stands on the concept that bringing the clock frequency f_{clock} to zero, set dynamic power to zero too.

2.1.2 Static Power Consumption

Static power is the power consumed when the device is powered up, but no signals inside the transistors of a circuit are changing value. CMOS technology is characterized by low static power consumption, and significant power is drawn only when the transistors in the CMOS device are switching between on and off states. Therefore, CMOS based devices do not generate as much waste heat as other circuit logic like TTL or nMOS.

Leakage current is the main reason for static power consumption in CMOS gates and can have four possible sources [23]:

Sub-threshold Leakage (I_{SUB}) is the current which flows from the drain to the source of a transistor working in the weak inversion region, it appears when the CMOS gate is not completely off, its value is give by the formula:

$$I_{SUB} = \mu C_{ox} V_{th}^2 \frac{W}{L} e^{\frac{V_{GS}-V_T}{nVth}} \tag{2.6}$$

where the equation's parameters represent:

- μ - the mobility of electrons in nMOS, and electron holes in pMOS

- C_{ox} - the capacity of the gate oxide
- V_{th} - the thermal voltage
- W and L - the dimensions of the transistor
- V_{GS} - the gate-source voltage
- V_T - the threshold voltage of the transistor
- n - a function of the fabrication process and ranges between 1.0 and 2.5

This equation indicates that this component of the leakage current depends exponentially on the difference between V_{GS} and V_T, hence, as V_{DD}[1] and V_T are scaled down to avoid dynamic power, leakage increases generating a trade-off.

Another problem that greatly complicates the design of low power systems is that Sub-threshold Leakage increases exponentially with temperature and, while leakage can be acceptable at room temperature, in critical environment or under not so uncommon situations, like a device being exposed to the sun, leakage may bring to miss design goals of the chip.

Gate Leakage (I_{GATE}) is the current which flows from the gate to the substrate through the oxide, it occurs because of tunneling current through gate oxide which, in 90nm and below productive process is just few atoms thick. Therefore, from the advent of 90nm technology, the gate leakage rose to about 1/3 of I_{SUB} and with lower processes like 65 and 45nm it can equal Sub-threshold leakage.

Gate Induced Drain Leakage (I_{GIDL}) is the current which flows from the drain to the substrate induced by a high field effect in the MOSFET drain caused by high V_{DG}.

Reverse Bias Junction Leakage (I_{REV}) is caused by minority carrier drift and generation of electron/hole pairs in the depletion regions

There are several approaches to face the problem of leakage current. One is the well known and highly used Multi-V_T [24] which consists in using high V_T cells with worse timing performances when design goals allow and low V_T cells when is necessary to meet timing constrains. Similar to clock gating for dynamic power reduction, power gating is another technique that consists in shutting down the power supply to blocks of logic that are not active, this avoids any leakage but is not drawbacks-free.

[1] V_{DD} is the supply voltage of the gate.

2.1.3 Some Observations on Power Reduction

Some conflicts could rises when the designer wants to reduce both static and dynamic power in CMOS technology. When V_{DD} is lowered in order to reduce dynamic power, I_{DS} is consequently reduced too, which results in slower chips. The drive current can be expressed as:

$$I_{DS} = \frac{\mu C_{ox}}{2} \frac{W}{L} (V_{GS} - V_T)^2 \tag{2.7}$$

Therefore, to maintain high performances while lowering the supply voltage, it is necessary to lower also the threshold voltage V_T as V_{DD} (and so V_{GS}) are lowered. Unfortunately, as we discussed in Par. 2.1.2 on page 35 lowering the threshold voltage results in an exponential increase in the sub-threshold leakage current (I_{SUB}) which is a great drawback as semiconductor technology is going from 90nm and below.

2.2 How to Reduce Power Consumptions?

Energy saving is the result of a series of solutions: some of these occur in the hardware design phase, involving architectural and technological solutions, while others may only be implemented when the final device is used and often need support of software to exploit best power saving techniques. As depicted in Fig. 2.1 at page 32, the mechanisms can be implemented at different abstraction levels: architectural, middleware and software.

Architectural Mechanisms

Architectural and technological mechanisms such as: clock-gating, multi-voltage and power-gating are at the base of almost all techniques for aggressive power management. Thus in this section it is worth to review the basic concept behind these mechanisms. In Fig. 2.3 is depicted how these mechanisms relate each other.

2.2.1 Clock Gating

Clock gating is a well-known technique to reduce dynamic power consumption. Since in a SoC, individual subsystems are used depending on the running application, not all the circuits are used all the time, giving space to opportunities to reduce power consumption. The distribution network of the clock in a chip can be a substantial percentage of total power of the SoC (30-35%, and up to 50% of the dynamic power) [25]. This is due to the fact that the clock signal has to be applied to most of the circuit blocks in the chip and it switches at every cycle.

Clock gating exploits these considerations and aims at disabling the clock to a portion of a circuit whenever that circuit is not used, preventing power dissipation

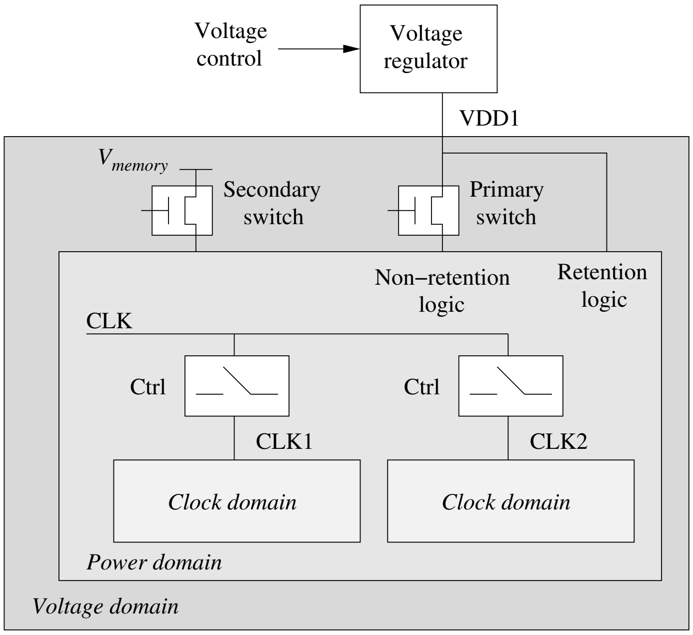


Figure 2.3: An overview of main architectural mechanisms, and their hierarchical relationship, that can be exploited for aggressive power management.

due to unnecessary charging and discharging of the unused circuits. Clock gating is usually applied by ANDing the clock with a gate-control signal.

The main challenge of this method stands in the detection of which circuits can ll not have transaction and thus can be gated, when and for how long [26]. If a circuit block is clock-gated while its functionalities are needed for the correct operation of the whole chip, an hardware error may occur; on the opposite, if the enabling signal (clock gating signal) is driven too frequently it may result in a higher power consumption than in the case the method is not applied to the circuit. Another aspect to be evaluated is the circuit area overhead which results by adding the necessary control components. In some cases it may happen that the introduction of the additional logic may overcome the benefits provided by its use. In these cases it is suggested to group the logic blocks with respect to their switching activity, in order to limit the circuit overhead.
An experiment [27, 23] produced some interesting results that suggest to use clock gating only on registers with a bit-width of at least three, since on lower width, the method is not silicon area efficient; moreover, the technique is more effective whether clock gating cells are placed early in the clock path.

Clock gating can be implemented not only at circuit level, during the phase of design and synthesis, but also at level of Operating System. In this case some drivers exports the software knobs that directly control the configuration of clock gates and are exported to a power management software or firmware. An example of this high level implementation of clock gating can be found in the Linux Clock framework, the in-kernel framework presented in Par. B.2.4 on page 145, which offers centralized control for all clock management related functionalities.

2.2.2 Multi Voltage

Lowering the supply voltage on selected blocks is a way to reduce power consumption caused by dynamic power and is the concept which hints multi voltage design techniques. The drawback of this method is that the delay of the gates increases as V_{DD} lowers, so one of the challenges on multi voltage design is to consider this trade-off and to find the best compromise.

In multi voltage design the internal logic of a microchip is partitioned in several blocks called *voltage regions* where, as the name suggests, each block has its own power rails characterized by a specific voltage. Voltage regions are also named *voltage domains* and it is common that in modern SoC the chip is divided in many subsystems with different goals, performance requirements and timing constraints. Then, this partitioning allows to apply dedicated supply rails to each block in order to satisfy its performance requirements and to observe its timing constraints, as a consequence of this fine tuning both static and dynamic power are commensurate with each single block and, considering the entire SoC, total dissipated power results significantly lower.
In Fig. 2.4 is reported as an example an overview of the voltage domains which are present on a commercial OMAP35xx SoC. This architecture define up to eight different voltage domain and every hardware block is composed by a logic block and an array block. This decomposition allow to host the bare combinatorial logic in a voltage domain that can be switched off once the device is not in use. To the contrary, the content of the array logic, which define the configuration of the device, is preserved by placing it on a separate voltage domain that is kept powered, perhaps exploiting some static power reduction technique such as reverse body biasing.

Although all premises let think multi voltage can be an effective way to reduce power consumption, it presents some important issues the designer has to cope with. First of all every block must be equipped with proper I/O pins and related power rails, so the complexity in designing the power grid increases, furthermore *level* shifters must be inserted in the chip to allow signals' driving among different power domains.

Level shifters are buffers that translate a signal from one voltage level to another in order to allow correct signal interpretation between two voltage regions [29]. It is clear that whether two connected power domains are respectively at 1V and at 3.3V, i.e. at radically different voltage levels, shifter are mandatory, at least to reach the transaction threshold. However, in modern chips, internal voltages are set around 1V, for instance the lowest voltage in a SoC block is 0.8V and the highest 1.2V, hence it may be not to immediate to understand the needed of level shifters, but

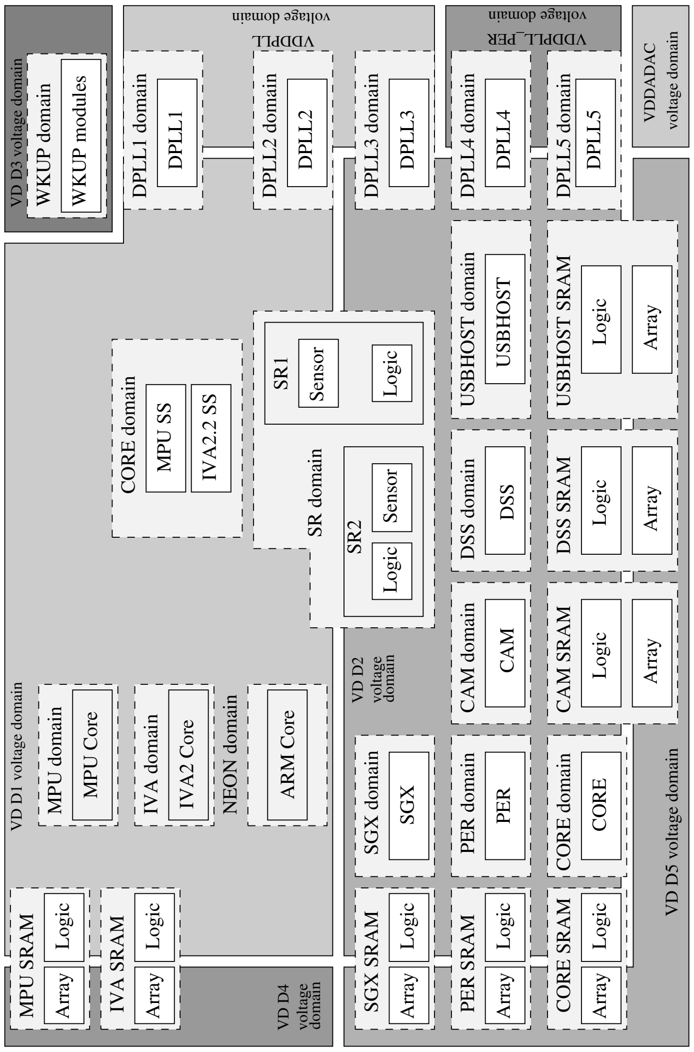


Figure 2.4: Overview of voltage domains in an OMAP35xx SoC. The device is split into eight voltage domains. This partition ensures independent voltage control of each voltage domain through dedicated SMPS or LDO. Functionally, however, voltage control of a voltage domain may depend on the voltage level of another voltage domain; for example, VDD1 voltage-level control may depend on the VDD2 voltage level. *Source: OMAP35xx Technical Reference Manual [28]*

if the 0.8V signal is used to drive a 1.2V gate, it impacts of both pull-up (pMOS) and pull-down (nMOS) networks generating crowbar currents that cause leakage. Usually level shifters are unidirectional, that means they translate either high to low or, in reverse, low to high voltages. Details about level shifter design can be found in [23].

Multi voltage strategies can be subdivided, with respect to the number of voltage levels that can be found inside each block of the SoC, in four main categories: static voltage scaling, multi-level voltage scaling, dynamic voltage scaling and adaptive voltage scaling.

Static Voltage Scaling (SVS) is used when different fixed supply voltages are assigned to different blocks of the chip [30]. This is the simplest design approach which does not pose particular matters with level shifting, but in modern complex microchips it does not guarantee optimal results as reported in [31].

Multi-level Voltage Scaling (MVS) can be considered an extension of SVS, where each subsystem of the SoC is supplied with a limited number of fixed and discrete voltage levels [23]. This approach allows to set different operating modes for each block and to save power by switching to the less power consuming mode, with lower voltage, when the block is idle or the workload is low.

Dynamic Voltage and Frequency Scaling (DVFS) is an evolution of MVS, where the fixed, discrete number of voltage levels is replaced by a larger number. In literature [32] [33] it is often found simply as Dynamic Voltage Scaling (DVS), but changing the voltage dynamically implies to change also clock speed as a consequence of the tight relation between the two dimensions which imposes a minimum voltage to guarantee that the transaction speed of logic gates can support the block operating frequency set by the clock signal. Therefore, voltage scaling acts in synergy with the frequency regulating the supplied power depending on the requirements of each block workload. Hybrid solutions are also supported and widely used: in a SoC, a subsystem like the host CPU may implement DVFS, while other blocks like the bus controller may exploit a MVS or SVS approach.
Since DVFS is the most effective technique to reduce power consumption in modern microchips, there exist several algorithms to implement DVFS which usually take advantage of software control.

Voltage scaling is done at circuit level via DC-DC converters which, taking an input voltage and a control signal, are able to output a set of different voltages which are then used to feed the SoC's subsystem. The switching time between a voltage level to another, called *voltage ramp* can be very slow (up to some milliseconds) and thus it could limit the frequency at which is possible to do voltage scaling. When frequency is scaled down the consequent voltage scaling can be performed immediately without any dependency, but in the opposite case the target frequency must be supported by the appropriate voltage level, hence voltage scaling must be performed before actually changing frequency with the resulting introduction of voltage ramp delay.

Frequency scaling is usually achieved via a Phased-Locked Loop (PLL) that, exploiting a reference clock source, scale up or down the frequency using multipliers and divisors for each controlled subsystem. If the target frequency of a subsystem requires a scaling-up, the DVFS is achieved through a procedure usually composed of the following steps:

1. the firmware or software, usually running at Operating System level, programs the new target frequency;
2. the processor programs the new supply voltage, and wait for it to be stable;
3. all clock dependent devices are suspended (e.g. self-refresh mode is enabled for external memories);
4. when the new voltage has been set by the power supply, the processor programs the new frequency;
5. if the new frequency requires just to change the multiplier/-divider value, this is accomplished and the subsystem can continue to operate without any pause;
6. if, instead, the new frequency requires a change of the PLL base frequency, the processor programs the PLL at the new frequency. In this case, clock signals mus be interrupted until the PLL is re-programmed to operate at the new frequency;
7. previously suspended device are now reconfigured and resumed;

The last step introduces a delay in the frequency scaling process that can be avoided inserting at least two multiplexed PLLs, in the way that while one is re-programmed to work at the new frequency, the other can continue to provide the correct clock signal without interrupting subsystems' operations.

The just discussed latency issues, suggests not to abuse of DVFS for two major reasons. First, at design time the set of discrete voltage/frequency pairs for each subsystem must be defined, this choice is not trivial and is a key point of the design. If the number of levels is high, it may happen that the system spends a lot of time choosing which is the best pair that satisfies the current power requirements, with the risk to cancel, or at least lower, the benefits provided by DVFS. Instead, if the number of levels is low, the voltage ramp delay increases, leading to overhead phenomena. Secondly, scaling voltage and frequency has the side effect of increasing the clock period since

$$T_{clock} = f_{clock}^{-1} \tag{2.8}$$

increasing the computing time independently by the number of clock cycles to execute a piece of code. This increase in the total time for processing impacts on the overall energy needed to complete a task which, as discussed in Section Sec. 2.1 on page 33 is strictly dependent on the time. The problem is especially critical when the chip is used in battery powered devices.

Adaptive Voltage Scaling (AVS) is an extension of DVFS which has been proposed as an effective power management technique where the system clock frequency and the supply voltage are dynamically adjusted, through a close-loop control, to meet the application throughput requirements [34]. The controller is usually a dedicated chip which necessarily embeds a performances and system state monitor, it is aware of the actual voltage delivered to different subsystems, it "knows" which portions of the silicon are slow and which are fast, it keeps track of temperatures in different blocks and consequently adapts voltage dynamically. The *Powerwise* technology [34] by National Semiconductor supports both closed-loop AVS and classic DVS. The Advanced Power Controller (APC), which design is described in [35], is a commercial chip supporting this technology.

2.2.3 Power Gating

Power gating is a technique to reduce leakage power when a subsystem is in sleep or standby mode [36]. It has been used since the advent of sub-90nm CMOS technology where leakage power became a major issues in the design of chips. The basic idea of such technique is to selectively power down some subsystems of a SoC when their functionality is not required for the correct operation of the system, while keeping other necessary blocks powered up [37].

Subsystems are then characterized by two power modes: an active mode and a sleep mode (low power mode) and the goal of power gating is to switch between these two states at the right time and in the correct manner to minimize power consumption with the lowest impact on performances and user experience. Comparing to clock gating, from an RTL design perspective, power gating affects inter-blocks communication and is a more invasive technique.

The simplest power gating design includes:

- *scheduler* controls the shutting down and power up procedure of each subsystem and can be a software component of the platform, usually programmed as a Operating System device driver or as part of the OS idle task, or in alternative, a dedicated hardware chip, usually devoted to manage power for the entire platform.
- *power gating controller* controls the CMOS switches that supply power to the gated subsystems.

- *power switching fabric* is a network of many CMOS switches distributed around the power gated subsystem with the objective of driving power to the block when active. It receive control signals by the power gating controller, which is, in turn, managed by the scheduler.

One of the challenges of this technique is that leakage power saving is not instantaneous and needs some time before reaching the target values, as depicted in Fig. 2.5. This happens because of the non-ideal nature of power gating technology and also as a result of temperature which introduces non-linearities in the switching process. This latency is critical not only from the performances point of view: the outputs of the power gated subsystem spend a lot of time at threshold voltage (V_T) causing short-circuit currents to the blocks of the SoC which are powered on and connected to the power gated subsystem. To avoid this phenomenon *isolation cells*, which are special CMOS cells immune to crowbar currents, are placed on the rail that interconnect a power gated with the powered-on block. Isolation cells are activated "on-demand", under instructions of the power gating controller.

State retention A major issue that is introduced by power gating techniques is the preservation of the internal state of a subsystem when it is powered off and the consequent restoration are wake-up. Preserving the state is important for some power gated blocks because it allows to save the time needed to reload registers from memory in case the state is not saved internally. Internal state saving and restoring is performed through different methods which include a software approach and a register-based approach.

The software approach consists in reading the state of a block before gating its power, saving the state in memory and when the block is powered-up again restoring data via a copy operation. This approach is slow and depends on the bus bandwidth and traffic, moreover conflicts on the bus may make state restoration non-deterministic and leave the woken-up block in a non operational state. Software is strictly platform dependent and lead to non-reusability and hard maintainability issues.

The register approach uses *retention registers*, which are special registers equipped with a shadow register that can preserve the register state during sleep time and restore it at wake-up. The shadow register is always powered by the supply voltage rail and is controlled with "save" and "restore", or alternatively to the formers a "retain" signal. The latter is an edge-sensitive signal that controls the operation of moving the state of the original register to the shadow register. The drawback of using retention register is the silicon area overhead that their use introduces, usually between 20% and 50%. A second disadvantage is the complexity added in the power controller design.

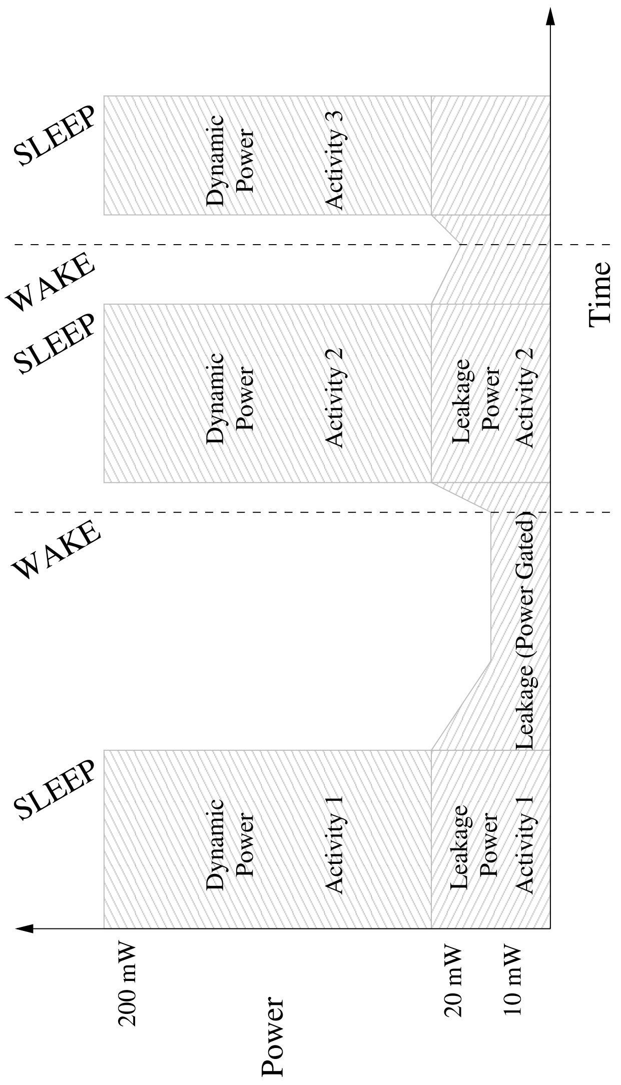


Figure 2.5: Realistic power consumption profile with power gating. The leakage power savings are not perfect and instantaneous; the full leakage power savings take some time to reach target levels. This is due partly to the (hotter) thermal profile of the preceding activity and partly to the non-ideal nature of the power-gating technology. Therefore the achievable savings are compromised to some extent. *Source: Low-Power Methodology Manual [23]*

Middleware Mechanisms

2.2.4 Resources hibernation (DPM)

Computer components consume power even when idle. As described in Par. 2.1.2 on page 35, the static power consumption can be reduced by properly tuning some voltages or even completely eliminated by switching off components during idle periods. Resource hibernation, frequently referenced also as Dynamic Power Management (DPM), exploits different hardware support, i.e. clock gating Par. 2.2.1 on page 37, multi-level voltage scaling Par. 2.2.2 on page 41 and power gating Par. 2.2.3 on page 43, to control power consumption through the implementation of *low power states* like sleep and suspend. In these particular states, the whole system or even just a subset of its components are totally or partially deactivated, eventually requiring, for some of them, the user interaction to bring back the system fully operational.

The ACPI specification [38] provides an open standard for unified operating system-centric device configuration and power management. This standard define the set requirements for the global states (G–States), the system states (S–States), the processor states (P–states) and the device states (D–states). While for each set of states, the state zero (e.g. P0) represent a working condition, all the remaining stats define a reduced functionality condition. The basic idea of these states is that the system (or the processor or a device), when it's not providing some service, can save power in several ways. Each of those ways has a different trade-off in terms of power-saving versus latency and performance. Thus, the higher the state number, the less amount of power consumed. While the power is reduced for higher number (deeper) power states, this power reduction comes at a price. The deeper the state, the longer it takes to leave the state, and the more energy this transition costs. In the light of these considerations, the smartest optimization approaches for resource hibernations can not ignore the concept of break-even time. The *break-even time* [39] is the amount of time a device must be in a lower power state, once entered, to effectively save some energy. This metrics is usually defined as:

$$t_{be} = \frac{E_{wake}}{P_{on} - P_{off}} \tag{2.9}$$

where E_{wake} is the awakening energy, P_{on} is the power consumption in the active state, and P_{off} is the power in the sleep state.

The decision that leads to a state transition has to be taken considering a simple set of metrics, for instance: the user inactivity on peripherals like keyboard and mouse in a PC or the touchscreen in a PDA, the pattern of device access requests issued by applications and so on. Disk drivers, networking cards and display, not only are some of the main power consuming devices in a computing system, but thanks to their common usage patterns are well interesting for the application of DPM optimizations. Different devices present unique challenges for the application of resource hibernation techniques.

In disk devices most of the power loss stems from the rotating platter. To save energy, an OS could spin-off the disk after a certain inactive period has expired and restart it during the next access. Of course, the decision of when to spin-down an idle disk involves a trade-off between performances and power saving. Network cards pose another challenge: since the naive solution to south down a network interface when not in use would possibly also disconnect the host from other devices, proper synchronization protocols are usually required to support proper communication among devices in a network. Applying DPM techniques to displays, which are usually the major source of power consumption, it is also interesting and could lead to significant reduction in energy consumption. Anyway also in this case the simple approach to dim a display after a sufficiently long interval of user inactivity may not coincide with user's intentions. Different approaches and corresponding heuristics have been proposed in past researches to successfully exploit resource hibernation on these kind of devices, I will review some of them in the following section.

2.2.5 Resources tuning (DVFS)

Modern computing systems, ranging from workstation multicore platforms down to embedded System-On-a-Chip based devices, provide advanced power management features to judiciously use energy. For instance, modern processors such as Intel's Atom and Core 2 Duo and ARM's Cortex A8 and A9 platforms, among others hardware technologies, incorporate also support for Dynamic Voltage and Frequency Scaling (DVFS). This technique allows to control the CPU operating frequency, by dynamically varying its speed according to the current workload, in order to reduce energy consumption during periods of low utilization [40]. All approaches essentially implement DVFS, to reduce energy consumption, by exploiting the non-linear relationship between the rate at which the CPU performs its works and the power required. Thus these techniques are addressed at reducing the dynamic power consumptions as defined in the previous section. Although a task will take longer to complete, the greater reduction in instantaneous energy consumption leads to an overall decrease of the total energy amount required to complete it.

Through this power optimization technique appears straightforward, serious real world complexities must be considered. A careful design is required to prevent the processor slowdown from degrading the user perceived applications responsiveness. From a theoretical standpoint we could minimize the energy consumption by setting the system service rate of the processor to be equal to the arrival rate of the new work. However this simple rule cannot be always applied: if we have only sporadic and short bursts of CPU load this simple approach could create very long response time for the user. Thus a perfect DVFS system must try to balance current demands with predicted future workloads. In general:

> the more the knowledge that the system has about acceptable service level and workload demand, the more the energy that it could reduce.

Two are the main complexities related to the implementation of DVFS techniques, the unpredictable nature of workloads and the indeterminism of real systems. Let me review them shortly.

Unpredictable nature of workloads - How to predict workload with reasonable accuracy is a first order problem that require to exactly know: what task will execute at which time and the amount of work required by this task. This problem is complicated by the fact that a task could be preempted (e.g. due to I/O operations) and also because of the well known 'Turing Haling Problem' [41] it is not always possible to predict the runtime execution time of an algorithm. There has been many efforts to solve this problem, especially in the context of real-time systems, where anyway only the worst-case execution time (WCET) could be [42, 43, 44, 45] accurately estimated. In real systems and with best-effort application the problem could not be efficiently solved because either the WCET is not the more probable case and also because micro-architectural innovations (e.g. out-of-order execution and hypertrading) make it difficult to apply statistical approaches.

Indeterminism of real systems - Real system's indeterminism make it difficult to determine the correct processor frequency even if we would be able to efficiently determine task workloads. Don't considering this indeterminism could lead to some risky misconceptions. First of all: it is *not* generally true that the *total system power* is quadratic in supply voltage. This is true for single transistors if we remain in the CMOS model context, but still there is not precise way to estimate the power dissipation of an entire system. If we consider modern multi-voltage domain designs, where the total dissipation is dominated by larger supply voltages (e.g. I/O banks), even if the smaller CPU's supply voltages are allowed to vary this will not reduce quadratically to overall system power [46, 47]. This lead to another common misconception: it is more power efficient to run a task at the minimum speed that grant to meets its deadlines. This is not always true since we must consider system-wide effects of DVFS: slowing down some applications could lead to keep active more time some peripherals thus consuming more power than that saved by the application [46]. And finally it is not exact to consider the execution time inverse proportional to the clock frequency. Indeed in instance DVFS could affect the way task are scheduled (e.g. due to preemption) and this could affect caches state end thus it could have side-effects on performances [48].

Software Mechanisms

Software mechanisms refer to the set of optimization that can be introduced into the code or performed by the code itself in order to optimize power consumptions. Thus, these mechanisms could be essentially classified as compiler based or application based. The former are optimization addressed by the compiler, either statically or at run-time, while the latter concern the software architecture itself. In this section it is worth to review the basic concept behind the compiler based optimizations. To the contrary, the application adaptation mechanisms are discussed in detail in the next section, which present the prior art, because these are the mechanisms and techniques of main interests for this research work.

2.2.6 Compiler based optimizations

Compilers can help on different ways to the reduction of power consumption, regardless of whether the target system support software–controlled power optimizations. Aside from the approaches to add code instrumentation that control the power mode of a system and its devices, they can also apply the common performance–optimization techniques to get benefits also from the perspective of energy saving, for instance by reducing the execution time. However, sometime performance optimizations increase code size and parallelism, thus augmenting the pressure on shared resources and peak power dissipation. Thus, even in the case of compiler based approaches, there is always the need to consider the trade–off between performance gain and power efficiency.
Basically the mechanisms available for compiler assisted power optimization can be traced to static compilation and dynamic compilation approaches.

Static compilation approaches could reduce energy consumption by optimizing the usage of some sensible resources. For instance the compiler could reduce the memory accesses by eliminating redundant load and store operations. Other optimizations could address the improvement of caches usage or perform aggressive register allocation in order to keep data as close as possible to the processor. A proper assignment of data memory locations and different loops transformations could help on improving the correct usage of the memory hierarchy to reduce power consumptions.

The main drawback of the static approaches is that the compiler has only a static view of the source code. Thus the compiler cannot exploit runtime informations on the execution context. Moreover, this kind of static compilers usually treats the source code as it is the only one running in the target. Instead, most of the real systems (excluding specialized embedded systems) are much more complex with many tasks running concurrently and competing for the available resources and with certain asynchronous events that introduce another level of indeterminism in the execution context. For example, context switches could have side effects on the memory hierarchy utilization regardless of the attempts of static compilers to

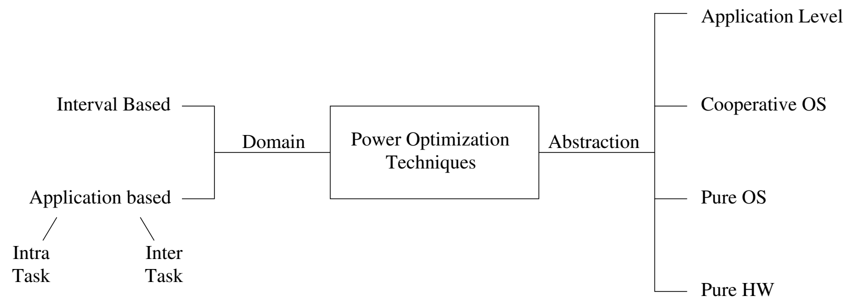


Figure 2.6: An high-level taxonomy of power optimization techniques. The main proposals that can be in literature can be classified according to two different perspective: the abstraction level or the domain of the optimization strategy.

improve its usage.

Some of these problems by dynamic compilation techniques, which are basically based on the exploitation of a feedback loop in order to optimize code at run–time according to the changes in the execution context. As the changes at runtime of some resources' usage could impact on the program behaviors, dynamic compilers are designed to recompile the program code in order to properly adapt to these changes. Of course, a dynamic compiler must be aware of the trade-off between the energy required for the recompilation and the energy saved after the optimization.

The dynamic compilation has been explored a lot in the past, mainly for optimizing performances. However, the continuous compilation can be exploited in a number of scenarios also to improve power efficiency. For example, monitoring the battery level a dynamic compiler could trade the data quality of processed data for a reduced power consumption by exchanging expensive floating point operations with integer ones. Beside the monitoring of some environment conditions, such as resource usage, there are other attempts that try to exploit some power model to address the compiler. For instance, is has been showed that a system-wide power model can be related to the usage of hardware event counters with a certain level of accuracy. Such a power model can be effectively exploited within a dynamic compiler to associate a power profile to the code and thus enable selectively at run-time different kind of optimizations.

2.3 Prior-Art Techniques

This section summarizes the state of the art for the research areas related with the main objectives of this thesis. It compares the most relevant works, also illustrating their advantages and shortcomings. The contents presentation follows the abstraction levels classification defined at the beginning of the chapter.

The power optimization techniques proposed so far in the literature can classified according to different perspectives, as depicted in Fig. 2.6. If we consider abstraction level they can be grouped into four main categories. In order of increasing application specific knowledge, and corresponding increasing performances, we have these approaches: pure hardware, pure operating system, cooperative application-OS and application level. Another possible classification could be provided considering the domain of the optimization strategy, in this case we have: interval based and application based techniques. Latters could be further subdivided in inter-task and intra-task techniques.
In the rest of this section, I'll consider the abstraction perspective to review in details some of the main past contributions that can be found in literature.

2.3.1 Pure Hardware Techniques

These techniques are based on specific hardware support, embedded within the processor, that measure the current CPU load and configure the processor frequency according to the inferred system utilization. One of the very first implementation of this approach has been the LongRun technology [49] used in the Crusoe processors [50] by Transmeta. This is a purely reactive and memoryless system belonging to the interval based methods: it simply measure the CPU load in a predefined time interval and scale the processor frequency according to the percentage of idle time. The final objective is to keep the CPU utilization near to 100%. Such methods are the simplest one to use because they don't require any modification of the operating system or to applications. A number of other commercial systems has been developed by main processor production companies.

The Intel's *Enhanced Speedstep Technology* [51] features some processor model specific register (MSR) that allow the software to influence the CPU clock which can transition between 6 different pairs of frequency and voltage settings. The Intel's Wireless Speedstep power manager [52] is an enhanced version of this technology which is embedded into PXA27x processors' family. In this case the processor's power modes are managed by an idle profiler and a performance profiler. The first monitor the idle thread while the latter exploit hardware performance counters to estimate how much memory bounded is the current workload. This is done by means of some statistics build around: caches usage, TLB misses, executed instructions and pipeline stalls.

The ARM's *Intelligent Energy Manager* (IEM) technology is a comprehensive hardware and software solution to scale core frequency a voltages which can be found on recent ARM cores [53]. This approach is based on an advanced hardware support provided by a dedicated SoC's embedded microcontroller, named Intelligent Energy Controller (IEC), and a three-level decision hierarchy of software policies. The IEC provide a closed-loop control on both processor power and performances, exploiting performance counters and internal/external SoC sensors. This hardware controller can be tuned by the hierarchical software policies: the bottom level provide workload estimation based on sliding window system monitoring, the top layer is specifically devoted to take care of interactive and multimedia applications requirements, and the middle layer allows other applications to communicate their workload requirements directly. Each policy tag its own performance prediction with a confidence rating which is used to compare decisions from different layers among them and decide the input to provide to the underlaying hardware controller.

The IEM technology is usually coupled with the *Powerwise* technology [34] by National Semiconductor, to support Adaptive Voltage Scaling (AVS). Conventional DVFS approaches determine the appropriate voltage for a clock frequency using a worst-case model with respect to the fabrication variations. As a result, the voltages chosen are often higher than they need to be. Powerwise gets around this problem by adding a feedback loop that continually monitors variations in temperature and other ambient effects through performance counters. These information are used to dynamically adjust the supply voltage for each clock frequency. This results in an additional energy savings, up to 45% [54], over conventional voltage scaling.

An online hardware approach for modern multiple clock-domain (MCD) microprocessors platforms has been proposed in [55]. The authors propose a formal analytic approach driven by dynamic workloads, where the MCD processor is modeled as a queue-domain network and the online DVFS as a feedback control problem with issue queue occupancies as feedback signals. The proposed online DVFS scheme, compared to prior approach, thanks to its automatic regulation ability seem to be more effective. Unfortunately the proposed DVFS schema has been evaluated only through a cycle-accurate simulation and, at best of my knowledge, no real-hardware implementations are known to exist.

Some main drawbacks of these techniques are:

- single system-wide frequency configuration: individual tasks do not have any direct control over the CPU power settings. Instead, a single CPU setting is determined, and thus it turns out to be typically based on the needs of the most resource hungry application. This could lead to sub-optimal configurations when a mix of applications is executing on the same processor or on multi-core systems where it's difficult to exploit optimal load balancing.

- estimation error: the OS needs to infer the processing needs of the applications by using on-line measurements and this could incur estimation errors.

- workload regularity hypothesis: it is very difficult, if not impossible, to predict irregular workloads using history information alone. Thus it is also difficult to achieve good results using only statistics from the operating system level when applications show bursty (unpredictable) behavior.

2.3.2 Pure OS-Based Techniques

These techniques try to improve the memoryless hardware approaches in two main directions: by exploiting OS scheduler knowledge and by allowing software system designer to compare different optimization policies.

Basically they are designed around the idea to determine a system-wide CPU frequency setting, based on the current task's processor demand. Workload informations, such as ready task's queue length and patterns of individual applications' I/O requests, can be easily collected at run-time by the scheduler and made available to a modular optimization policy, along with hardware performance counters too. Optimization policies basically exploit these data to identify how much busy is the processor, over a predefined time interval, and than estimate how much busy it will be in the next time frame to adjust accordingly the CPU speed. Thus these methods, as long as the hardware's one, could be classified as interval-based.

Different optimization policies has been developed, which basically differs on the amount of informations exploited and consequently on the way they estimate future workload. The earliest algorithm developed in this class was *PAST* [56] which simply define an hysteresis cycle based on the CPU idle time: if the processor idles longer than a lower-threshold, its speed will be decreased, otherwise, when it remains busy longer than an upper-threshold, its speed will be increased. This policy is pretty simple and, even if it is error prone since it makes decisions based only on the most recent informations, is also very reactive and with few variations and a more parametric approach is the default one used in recent Linux kernel by the CPUFreq framework [12]. A number of algorithms has been developed to extend *PAST* and better exploit available informations in order to save more energy. The AVG_n [57] approach in instance consider many measurements collected over a larger time window and estimate the CPU usage for the next interval as a weighted average of usage measures in a certain amount of previous intervals.

In [58, 59] is presented Vertigo: a power management extension for Linux. This framework also makes its decisions automatically, without any application-specific involvement, but exploit a hierarchy of workload's specialized performance-setting algorithms. Each algorithms operate independently from one another and has a set of specific configuration parameters. The main goal of this framework is to transparently perform performance reduction without causing the software to miss its deadlines.

The Processor Acceleration to Conserve Energy (PACE) algorithm [60, 61] is a scheduler modification which increase the CPU speed as a task progress if deadline missing becomes more likely. This technique depends on the capability to estimate the probability distribution for the work requirements of a task. The authors show how is possible to make such an estimation and also how it is possible to approximate the perfect schedule with one that limits the number of frequency changes in order to reduce run-time overhead while still getting substantial CPU energy saving.

A number of works has focused on DPM techniques, based on pure OS solutions, for the power optimization of peripherals, mainly: disks, network cards and displays. Disks optimizations are usually based on *Adaptive Dynamic Threshold Adjustment*, a technique to adjust at runtime the idleness threshold before a idle disk can spun down. Based on this technique, in [62] is presented an approach to cluster disk access requests, thereby lengthening idle periods. To create opportunities for clustering this algorithm adopt a double strategy of both delaying non-urgent disk requests and aggressively pre-fetching disk data into memory. In large scale server, with continuously processing workload, disks idle periods are almost absent and cannot be successfully exploited to spin down disk. In this specific context an interesting approach has been proposed in [63] where is investigated an alternative optimization strategy based on dynamic RPM control (DPRM). The authors propose to modulate the plate rotational speed according to the workload demand. An adaptive threshold algorithm is at the base of another proposal [64] for power optimization of wireless network cards. In this approach the authors define a heuristic that use timers to track idleness of devices. An idle timer is used to enter the device into listening or sleep mode for a run-time tuned hibernation time. When network activity is detected, the algorithm decrease the hibernation time acceptable, which instead is increased on idle periods detection.

The *Homogeneous Architecture for Power Policy Integration* (HAPPI) [65], is one of the most recent works on OS level DPM targeting multiple devices. This proposal is interesting because, starting from the evidence that a policy can outperform another under different conditions, tackle the problem of finding the "best" policy for all systems. Thus the work advances the proposal for an OS architecture that support: the integration of multiple policies, their simultaneously run-time comparison and the independent selection of the best one for each specific controlled device without user or administrator intervention. The work show how different predefined and well-known DPM policies can be effectively switched at run-time to better control each device based on workload demand.

Some main drawbacks of these techniques are:

- run-time overheads: the software control is more flexible but require some work to be performed by the optimization policy, thus the energy consumption vs benefits tread–off must be carefully evaluated and considered.
- no direct input from applications: these techniques infer the control actions from the analysis of the operating system state only. Thus, only indirect informations from the applications can be exploited; for instance: if an application open a device then it is highly probable it will be used. However, even if these approaches are not purely re-active the analysis of the system state and its processing require some work that should be considered in the evaluation of the control trade-off.
- disjoint policies proliferation: within the operating system we could find gen-

erally multiple policies for the optimization of different subsystems. Unfortunately, it is not possible to grant a system-wide optimization considering only the overlapping of multiple and disjoint controllers. Indeed, the most promising approaches within this class propose the design of a certain kind of centralized controller.

- complexity of centralized controllers: the centralized supervisor based approaches are usually based on a modeling framework which is more complex than a system integrator would like to see. Thus, in practice these approaches have showed very limited applicability in real products.

2.3.3 Cooperative-OS Techniques

In cooperative approaches the Operating System try to exploit some domain-specific informations communicated directly by the applications. These "application hints" increase event further the level of knowledge about tasks' requirements and optimization algorithms could exploit them to better identify optimal CPU frequency configurations.

The Milly Watt Project [66] was one of the early attempts to explore the definition of a power-based API supporting cooperation among applications and the operating system on setting an energy-use policy. One of the main contribution of this work was the focus posed on the need to raise the importance of the energy consumptions reduction, among performance goals on designing new software.

In the context of this project project was also the definition of ECOSystem [67]: a prototype energy-centric OS. This framework defined a powerful mechanism to formulate energy goals, based on a "currency model" abstraction [68], to model power resources as a monetary unit. Essentially different system resources has a cost to be used and applications must pay to use them. The ECOSystem operating system periodically distribute currency to applications according to the power optimization goals. This was an interesting example of a powerful mechanism to formulate energy goals and to unify resource management policies across diverse competing applications and spanning device components with very different power characteristics.

Many other approaches proposed the usage of a proper API to exchange informations from user-space to kernel. In [69] a simple two layer software architecture is proposed. In [70] a more complex interface is designed where applications are allowed to chose when access I/O devices, based on their relative cost, while still hiding devices' details. While giving applications a fine-grained control on devices, this approach is particularly interesting because of the introduction of "ghost hots" mechanism that allow devices to learn when applications require them and adapt consequently their own optimization policies. A similar work is proposed in [71] where a compilation technique is used instead. Suitable application's code transformations are operated in order to cluster device accesses, in instance to increasing the opportunities for spinning down a disk.

In [72] is proposed an integrated power management approach that unifies low level architectural optimizations (CPU, memory, register), OS power-saving mechanisms (Dynamic Voltage Scaling) and adaptive middleware techniques (admission control, optimal transcoding, network traffic regulation).

The approach proposed in [73] require that applications are power-aware and specify their average execution time (AET) and the deadline to the scheduler. This approach define an energy priority scheduling (EPS) algorithm to support such power-aware applications. The scheduler orders tasks according to their deadlines and tasks overlapping level. Indeed this algorithm does not always yield the optimal schedule, it has a very low complexity and for very bursty applications show processor power consumption reduction up to 50% without missing any specified deadline.

Some main drawbacks of these techniques are:

- application modification: it is generally required to integrate the applications within the framework by modifying them in order to pass the required informations to the in-kernel framework. Unfortunately, even when this is possible, generally it is not always something welcomed by developers which prefer to focus on application functionalities instead to power management stuff.
- reduced application portability: modifying an application to integrate on a framework could impact on its portability since it is not conceivable that all systems use the same framework.[2]
- communication overhead: inevitably the information exchange between the OS and the applications introduce some overheads in the control policy. Moreover, faulty applications or even worse malicious ones can interfere with the control policy conditioning its effectiveness.

2.3.4 Application-Level Techniques

Rather than a partnership between the OS and the applications, these techniques try to exports the entire burden of power management to the user level. The basic idea it that: since applications best know their processing requirement, we should allow them to make decisions on power management. The OS's role, on these approaches, it is solely that to enforce protection and isolates applications from the power settings of other applications. Thus these approaches resembles mainly the philosophy of the Exokernels[3] [74]. Since the Exokernel project successfully demonstrates the benefits of application-level networking, memory management,

[2]Unless it is a well defined standard, such as ACPI. But up to know this kind of standard has been defined only for lower abstraction levels concerning the OS interfaces towards the hardware.

[3]An Operating System architecture where the kernel grants complete control of various resources to the applications and only enforces protection to prevent applications from harming one another

file systems, and CPU scheduling[75], these approaches try to extends this notion to application-level power management too.

A low-power application design methodology is proposed in [76]. This approach adopt an "architecture-centric" view, where each application can be decomposed into fundamental elements (e.g. process, communication mechanisms, handlers). Thus, an application can be represented as a software architecture graph (SAG), and power consumption is related to interactions among its basic components. The authors describe than a simulator based estimation of basic application power consumption. This estimation is used to compare energy consumptions when the application is modified by transformations operated on the SAG, such as: merging processes to reduce communication energy or redistributing computation among processes. A greedy approach is proposed for the exploration of SAG transformations, where they keep applying modifications until no more energy could be saved or no more known transformations exists.

A number of work has been done on the field of application-controlled DVFS techniques for multimedia application.Video decoding has been the reference use-case of these techniques, which relay both on off-line estimation of CPU decoding's demands [77], or its on-line estimation [78, 79, 80, 81, 82].

The approach proposed with Chameleon [83] is an application-level power management architecture to support applications embedding power management policies. The authors argue that applications best know their resource and energy needs and consequently they should be able to define better power management policies too. In a Chameleon enabled kernel, a complete control over the CPU power settings is given to the power-applications, which are allowed to specify their CPU power setting independently of each another, while the OS enforce only isolation between an application and the settings used by another one. The authors show also how simple it could be to develop effective application-level power management policies for commonly used applications belonging to different application classes such as soft real-time, interactive and batch.

Many other application level techniques belong to the class of application adaption, where power consumption is treaded with quality or data fidelity. In [84] a video encoder is presented which can tune its compression efficiency by trading computations' accuracy for reduced energy consumption. This is also an example of mixed hardware and software approach because the solution is based on two algorithms that work side-by-side: one exploit DVFS support to tune hardware parameters, while the other tune the parameters of the video encoder. Other techniques for application adaption of video applications are presented in [85, 86], while in [87] and [88] similar techniques for application adaption are applied in the office application's context.

The Odyssey operating system [89, 90] is another example of OS controlled application adaptation. This framework, especially designed for multimedia and WEB applications, monitor resource usage and notify applications when they fall below the required level. In turn applications lower the required quality of service until resources are still available.

An experimental face recognition system, Face-off [91], has been proposed for the DPM control of displays. This application level approach is based on an algorithm of face recognition running periodically on a snapshot of the monitor's perspective. This allows to better capture user intentions but a drawback is the repetitive polling for face detection. The usage of a proximity sensor is foreseen as an interesting optimization to trigger the face recognition algorithm only when user is detected.

The main drawback of these techniques is:

- application specific: the implementation of a power control in user-space can be approached only considering application specific contexts. Otherwise, the complexity required to make cooperating many different applications, considering also the protection mechanisms enforced by the operating system, makes impossible to implement effective solutions.

2.3.5 Cross-Layer Techniques

As noticed in the introduction of this chapter, power consumptions depends on decisions that span all layers from transistors up to applications. The development of holistic approaches, that aggregate data from multiple layers into power management decisions, is a popular research topic. Indeed, a number of approaches based on cross-layer adaptations have already been proposed.

Forge [72] is an infrastructure based solution, targeting power optimization for networked multimedia applications. The proposed multi-layer architecture consist of an hardware layer with a set of tuning parameters, an operating system and compiler middle-layer and a distributed middleware in the upper layer supporting running applications. The proposed architecture integrate multiple levels of adaptations. The hardware layer support DVFS and DPM techniques as well as a set of architectural tuning parameters. Local middleware monitor available resources and send informations to the centralized portion of the middleware running on a proxy. Remote off-line profiling, based on simulation, along with the collected informations allows the proxy's middleware to tune the served data stream.

Another cross-layer approach targeting the power optimization of mobile devices that primarily run multimedia applications is the Grace OS [92] project. This is an attempt to integrate DVFS, power-aware task scheduling and QoS demands for the power optimization. Even more interesting this framework is based on a hierarchical two layers of adaptations. A global policy act as a central coordinator that monitors resources availability and respond on widespread changes. On the other hand, local policies respond to smaller workload variations for resource usage fine tuning. The proposed solution is based on three local policies controlling CPU frequency, task scheduling and QoS parameters. Unfortunately, even being effectively implemented into a Linux kernel by extending the traditional real-time scheduling, this framework has been designed only for the energy optimization of real-time multimedia tasks with fixed periods and deadlines.

Another multilayer framework for multimedia applications power optimization is presented in [93]. The main innovation is a middleware layer in user-space that perform admission control on application and their QoS demands. This control is supported by run-time informations collected via an OS interface and by meta-information that each application entering the system should provide to the middleware. These meta-informations comprise QoS levels accepted by the application and corresponding power consumptions profiles.

In [94] is explored the possibility to exploit compilers and OS cooperation to save energy. Still the target is on real-time applications with predefined dead-line and worst case execution time (WCET). The compiler instruments the code adding WCET informations that can be used run-time by the OS to perform DVFS.
A compilers based approach has been proposed also for the DPM of wireless network cards [95]. This technique target a simplified scenario where a device has its virtual memory on a proxy server, and page faults are exploited to trigger the wakeup of the network interface. The compiler is used to identify program regions, with an array access patterns to memory, and simulating a least-recently-used memory access algorithm it can understand where to instrument the code with the proper control network interface status.

These are for sure the most promising techniques for implementing a power optimization framework which satisfy all the requirements discussed in Sec. 1.2 on page 12. However, the solutions proposed in literature show still some limitations, most notably:

- application modification: to better take advantages from an holistic approach, the compiler support is required on some proposals in order to instrument the application code. The instrumentation allows to collect better and more complete informations at run-time about the application requirements.
- reduced scalability: no frameworks has been proposed which considered the scalability of the control as one of the main requirements. The scalability can be compromised by both: the complexity to support the framework, for instance on a general purpose platform, and the run-time overhead increase due to the increasing number of components being added to the system.
- limited application: even the frameworks which has had a real implementation, on used operating system such as Linux, are limited to specific and well defined working contexts. This is mainly due to the complexity of the approach proposed and it is a consequence of the previous point too.

Technique	Main Advantages	Main Drawbacks	Example
Pure–HW	- low run-time overhead; - no SW modification;	- pure reactive; - memoryless; - no task aware;	LongRun Technology [49], Speedstep Technology[51], IEM [53];
Pure–OS	- task view; - exploit OS state; - no SW modifications; - many policies;	- some run-time overheads; - no task input; - disjoint policies;	CPUfreq [12], Vertigo [58], HAPPI [65];
Cooperative–OS	- task knowledge; - application feedbacks; - resource management;	- application modifications; - communication overheads;	Milly Watt Project [66], ECOSystem [68];
Application Level	- simple kernel support; - better task knowledge;	- offline processing required; - specific application context;	Chameleon [83];
Cross–Layer	- holistic approach; - centralized control middleware;	- limited exploration; - offline application analysis;	Grace OS [92];

Table 2.1: A summary of pro and cons of the main techniques for power management.

Chapter 3

An Instance of the Technique

"True optimization is the revolutionary contribution of modern research to decision processes. "
George Bernhard Dantzig

In the overview I introduced a fundamental approach to construct distributed, fine-grained, dynamic and fast techniques, to support both power consumptions and performances estimations of embedded systems when running on given use cases. This fundamental approach may derive many techniques, depending on which modeling choices are taken. This chapter presents one instance of these possible techniques, which is especially suited for multimedia mobile embedded systems.

This chapter discusses the models on which this technique relies, the basic steps which compose it, the activities which it involves and the people which are supposed to carry them out.

3.1 Focusing the Reality

We are interested in controlling the energy-performance trade-off of SoC based mobile platforms. These systems are composed of several perhaps complex devices[1], some internal and others external of the SoC, that interact by exchanging informations and data to provide some kind of service. Operating conditions evolve over

[1]In this part of the work we call *device* a component of the system: a device can be a CPU, the main memory, DMA controllers, network interfaces, some storage, etc. The reader should not confuse this meaning with the notion of device as that of a product, e.g a mobile phone or a GPS navigation system.

time and user required services can be different time-to-time, thus devices usage and their configuration must be changed accordingly. Usually the entity in charge to manage devices is the Operating System (OS), more precisely device drivers, that have the necessary knowledge to understand how to reconfigure devices based on the required service to support.

Ensuring a proper energy-performance trade-off on such complex systems is a challenging goal. The great number of devices and related control parameters, along with the variability of application requirements in terms of Quality of Services (QoS), makes it difficult to understand how to define a suitable system-wide set-point.

Finite discrete parameters

The state of a device can be described by its configuration. This configuration defines how the device behaves: what services it provides as well as how they are performed, for instance in terms of achievable performances and power consumptions. Therefore, controlling the state of a device thus requires to identify a suitable configuration to achieve expected system behaviors.

A device configuration is described by a set of registers that allow to define the values of its *tunable parameters*. The device is the object to be controlled and can be associated to the target system, tuning parameters are the actual control points we can use in order to provide a control input to a device.

It is worth noticing that these tuning parameters can accept a finite number of possible values and, for the specific devices we are considering, control input is represented only by discrete values. So, control inputs of systems we are interested in, will be represented by a *finite number of discrete values*. Device drivers know exactly the set of values that a control input can accept and how these affect device behaviors. This is an important property of the systems of our interest and should be considered to properly design the controller.

Discrete event system (DES)

The state of a target system is defined by the values of its state variables. This state may generally be not directly observable, but in our specific application context, where the system is constituted by a collection of devices, this state is known in every moment of the evolution of the system and corresponds to the collection of device's configuration registers.

Operating conditions of the entire system can change with time, either due to user interaction, running different applications or requiring different services (e.g. switching from loudspeaker to earphone), or to mutating environmental conditions such as different network access channels availability. Therefore these changes in operating conditions can be considered as *asynchronous discrete events* over time.

When such an event occurs, usually a system reconfiguration is required to better support the new status. A system reconfiguration typically involves a change of the internal settings of one or more devices in order to support the requested service while optimizing the performance and energy efficiency trade-off. Hence, according to the discussion on Par. 3.1 on page 62, each reconfiguration will move the system to a new *discrete state* defined by the contents of all of configuration registers.

From the above considerations we can observe that our application scenario is a *discrete-state* and *event-driven* system (i.e. *discrete event system)*, since the state evolution depends entirely on the occurrence of (perhaps asynchronous) discrete events over time. The design of a controller for such kind of systems is supported by the theory of discrete control [96].

Configuration overheads and benefits

Changing the state of a device requires some processing time, not only to execute the software routine that effectively reconfigures its registers, but also to understand what are the correct values to load in a new configuration. This time is largely considered an *overhead* since it requires some power without producing useful work, but only to move the system to a possibly better operating state.

Since reconfiguring a system is a resource consuming task, in order to have benefits from a system reconfiguration it is generally required that the system will remain in the new state for a minimum time amount of time. This is especially true if we consider power optimizations: when the system is moved to a lower-consuming state, since energy is spent to reach this state, in order to have benefits the system should remain in such state at least for the time necessary to compensate the energy consumed to activate it.

The design of the controller of system configuration should consider these aspects and take into account both overheads and benefits associated to each device in order to understand when a reconfiguration is feasible and convenient according to power and performances requirements.

3.2 Abstracting the Reality, Modeling the Abstraction

The objective of this technique is to provide support for system-wide power and resource management policies. The real path which leads from user-space QoS requirements down to an optimized system-wide configuration, in therms of energy reduction and performances tuning, involves a large number of steps. Many of these steps show great complexity, as I detail below. For reasons of effort, performance and generality, it is not convenient to account completely and exactly for this complexity at all the abstraction layers. Instead, a convenient trade-off between effort, performance, generality and accuracy should be chosen.
For example, a specific's device hardware capability should be modeled exactly when the effort is acceptable, it leads to a model which is general enough to be easily tuned and used on different systems, it has acceptable run-time overhead

and it leads to a significant increase in optimization accuracy. Otherwise, it is more convenient to model its behavior in a statistically-consistent way. It may be a reasonable idea to even completely neglect the effect of a specific hardware feature, if the incremental added accuracy is negligible.

In the light of the above considerations, this section discusses how to obtain a model of a real system and the corresponding power vs performances trade-off problem, which presents an acceptable trade-off among the accuracy it can provide, how easily can be generalized, the design effort it involves and the run-time overhead it causes. I derive my technique by abstracting the platform and obtaining an abstract description of its capabilities and resources. Then I model this abstract description to obtain a solution for the power and performances optimization problem. The two terms "abstracting" and "modeling" are used in this context in the following sense:

- *abstracting* an object means replacing it with a simpler object, which is functionally equivalent and numerically consistent (either exactly or statistically) with the original one, but it exhibits a behavior which is easier to model. An abstraction is an operation which reduces complexity. For example, abstracting a real network card device may means describing it as a simpler virtual network card, such that the real and the abstract devices exhibit statistically-similar working modes measured in activation latencies, resources utilization and power consumption. The original object and its abstraction are functionally equivalent: in the example, both support the same working mode, both exhibit a deterministic behaviors. Abstracting some working mode of a device, for example the connection speed of the network interface, may mean replacing it with another description where some details (e.g. the required frequency by the radio-cell's clock) are not considered, or described at an higher abstraction level (e.g. the network connection standard used). As just suggested, introducing an abstraction at a given stage induces a simplification not only on that stage, but also on its output information and, consequently, on the following stages in the flow. This means that not only behaviors but also information flows are subject to abstraction, and that abstraction of objects which belong to the same information-flow chain must be consistent with each other. Device drivers already provide an abstraction level within an OS. We need to identify how to eventually extend this abstraction so that to be able to provide consistent informations to the up-standing optimization framework;

- *modeling* an object which takes some input and yields some output means replacing it with a function which estimates the cost (in a general sense) of the output from the cost of the input. Modeling is an operation which replaces a behavior with a function which accounts for the cost of that behavior. The model of an object is not functionally equivalent to the original object: a real device works in a certain configuration using some resources and consuming some energy, whereas the model of a device yields an estimate of the working mode's energy consumption and resources requirement. An application require some QoS to the OS, whereas the model of the system yields an estimate of the cost of executing that application on different configurations;

Here I examine a real embedded system architecture, and I define how is possible to give an abstract representation of its components which can be properly used for our optimization problem. Finally, I present a model which allows to determine the optimal system-wide configuration, considering both to available resources and required QoS, based on a configurable global optimization policy. Figure 3.1 represent the details of where we apply abstractions and where the model lays. I will not implement all the abstraction of a real system, since it would require the complete definition (down to drivers level) of a platform, its devices, and the development of associated drivers and local optimization policies. Such an implementation is not useful for the purposed of this research. Instead, I do implement the system-wide optimization model. This model is the fundamental goal of this thesis.

3.3 From Reality to the Abstract Layer

3.3.1 Resources Abstraction

A modern computing system comprises different resources at different abstraction levels. Resources can be both hardware defined, such as a quantity of memory or the throughput of a bus, or can have a more software-related meaning, such as the system latency or the type of audio codec used by an application. Whatever its nature is, generally a resource is provided by someone, in our context it is usually an hardware device, and required or expected by another entity, either a software application or another device. Requiring and correspondingly providing a certain resource is strictly related to the Quality-of-Service (QoS) an application can offer: quite often it happens that the better the quality levels required is the better the performances we get but also energy consumptions increases, while reducing the QoS level preserve energy with a corresponding performances degradation.

From the above considerations it is clear that considering resources is important to tackle the problem of power and performances optimization. Since resources can have different nature and representation it is convenient to abstract and represent them use a single concept. This abstraction if the first one I introduce and should be sufficiently general to represent any kind of hardware or software resource and to track their availability and requests. Even if not all resource can be represented by a continuous range of values and thus measured, surely at lease it is possible enumerate them. In instance the audio-codecs required by an application or provided by an hardware accelerator can be represented by a finite discrete enumeration. For this reason I adopted the term "metrics" to refer to them, more precisely I define:

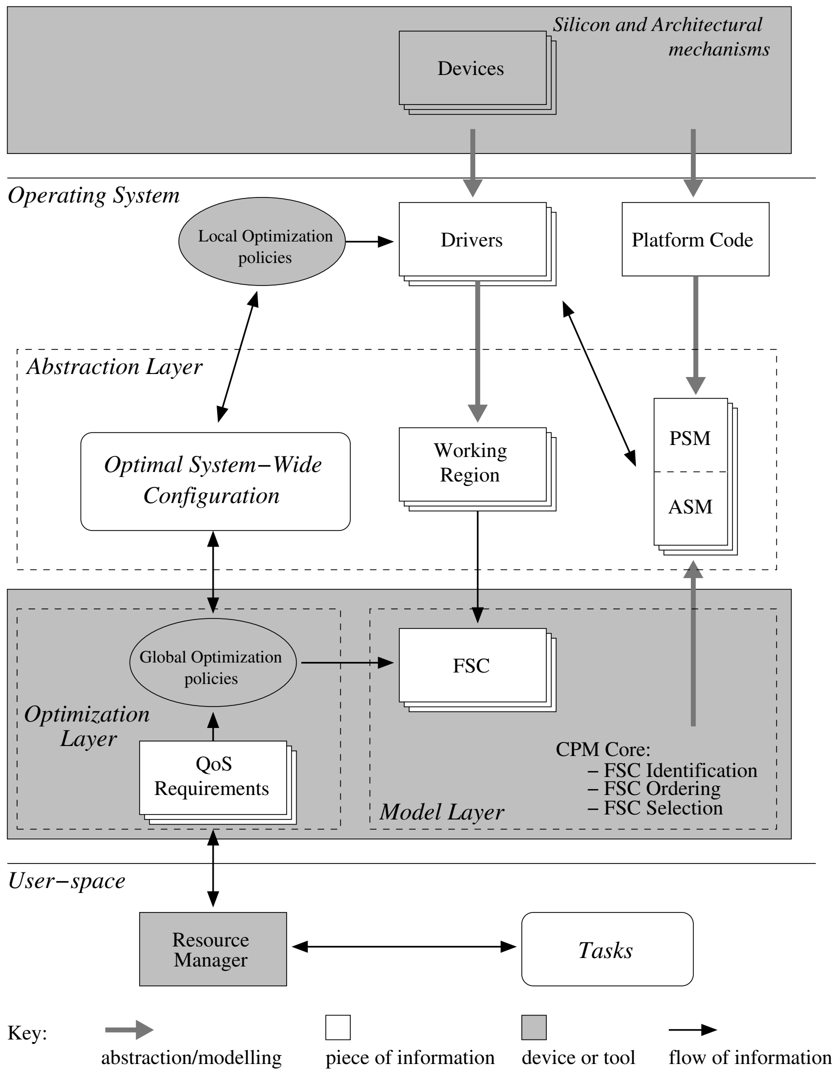


Figure 3.1: A real computing system, with many different devices and applications, is so complex that it is not convenient to model all this complexity in order to solve the consumption and performances optimization problem. Therefore I perform an abstraction and modeling (see Sec. 3.2 on page 63). This figure represent where abstraction and modeling are done.

> *System-Wide Metric (SWM)* - an abstract information shared among all entities of a system, either devices' drivers and applications. It represents a "system's resource", which usually has a provider and one or more clients, and it is used as a "global parameter" of the optimization framework proposed. A minimum and maximum value are associated to each SWM defining a range over which the corresponding parameter can take either continuous or discrete values.

In this definition I use "system's resource" based on the quite general meaning provided before, i.e. a resource can be both an hardware or software-related meaning. The "global parameter" refers instead to the independence that such metrics have with respect to both their clients and provides, i.e. there are not SWM which are specific to a device or an applications, conversely they can be referred by any other system entity.

Ad I will better explain on the following sections, these entities are used to provide a complete and consistent description of the whole system's requirements that could be deduced both from device's drivers and from informations grabbed from running applications.
Example of SWMs are: the latency of the whole system, the bandwidth of a communication channel, the power state of the device, the environmental conditions coming by sensors devices, etc.

A SWM can be classified with respect to two attributes which allow to better define some propertied of the abstraction they represent:

Composition: define how multiple requests on a SWM should be aggregated. Since a SWM represent a resource it can have multiple clients, and thus it becomes particularly important to define how to handle multiple requests of the same resource. This is a typical resource management problem: a system-wide metric represents a resource and quotes of it can be assigned to one or more clients. To grant an always updated view of resources already reserved and resources still available: multiple requests issued on the same resource should be properly aggregated. This is where the composition attribute of an SWM is used to understand how to properly aggregate multiple requests.
A SWM is defined to have an *additive* composition when a new request asserted on it should be composed to the current value via an addition. This is the case of shared resources like bandwidths and throughputs where the composition of a new value requested on them cannot ignore the current situation.
In instance, considering am hardware communication channel, if the system is already using a certain amount of bandwidth on the channel and an application express its own requirement of bandwidth, this request must be composed starting from the current utilization level of the channel and sum the new request to that level. This assumption allows to keep the system in a

safe situation by setting the conditions to be able to react when a requests on an SWM would cross the resource's limit values.

A SWM is defined to have a *restrictive* composition when a new request asserted on it should be composed to the current value via comparison function, i.e. maximum od minimum, eventually overwriting the current value of the SWM with the new one.
his is the case of system-wide parameters like latency: if an entity requires a certain system latency the value that should result after the composition is given by the minimum between the current latency value and the requested one. Hence, the aggregation is not performed via an addition but with an overwriting of the old value.

Type: defines how a SWM behaves with respect to the corresponding QoS. I already described at the beginning of this section how resources are strictly related to QoS. The quantity of a certain resource, and thus the value of the SWM abstracting that resource, has a certain relation with the corresponding quality level. In instance a low-latency system is defined to be more responsive to interrupt that a system with larger latencies. Thus, considering the 'system latency' a metrics of QoS, in this specific case the first system is better that the latter. Conversely, if we consider 'power consumption' as a QoS metrics and in the previous example we experiment that the low-latency system is more power hungry than the second, than from the perspective of the 'power consumption' metrics the second system is the best one.

A SWM is defined to have *Greater is Better (GiB)* type when higher values of its parameter correspond to an increased QoS. In instance, the bandwidth on a network interface is GiB because as much as bandwidth is available, as much as QoS is achievable from the user perspective.
A SWM is defined to have *Lower is Better (LiB)* type when smaller values of its parameter correspond to an increased QoS. In instance, the latency on a memory transfer is LiB because a lower latency allows to have better performances.

3.3.2 Architecture Abstraction

In the definition of SWM I used the attributes 'global' to stress the independence that such metrics should have with respect to any specific applications and hardware. To better address both the 'scalable' and the 'fine-details' requirements of modern computing system (Sec. 1.5 on page 19) I refined the concept of SWM to allow the abstraction of platform and architecture specific metrics while still not compromising the portability of the final solution proposed by the technique I defined. To achieve this result, the framework I propose support two classes of system-wide metrics:

Abstract System-wide Metrics (ASM): are SWM completely platform independent. Metrics belonging to this class grant portability to the proposed solution because, begin present in any implementation of the framework, can be used by any application regardless of the actual hardware platform they will run on. This metrics should be sufficiently abstract and thus they can be used to represent high-level of abstraction resources, e.g. the network bandwidth or connection mode, the backlight power-saving mode, the expected memory usage, etc.

Platform System-wide Metrics (PSM): are SWM that could be added to the framework to better model some platform-specific resource. Differently from the previous metrics, the ones belonging to this class cannot be used by applications because of the reduce portability. Anyhow they are available within the platform specific components of the framework. This allows to easily take advantages from the framework provided interfaces and optimization strategies wherever it is considered profitable to improve the framework precision.

It is worth to notice that this to classes of metrics are functionally equivalent from the perspective of the framework and the how it handle them for the optimization purposes. The only significant difference is on who can access ad interact with them. In the case of PSM this is reserved to some platform-specific code (either in kernel or user-space), while ASM can be used by any entity and are especially useful to build a really portable solution since they define a framework's API which is completely platform independent. In the following sections I will explain also how this two classes of metrics can be better exploited to built both 'scalable' and 'fine-detail' optimization solutions, by properly defining relationships between metrics belonging to the two different classes.

The collection of SWM available on a system, both ASM and PSM, do not define only a convenient abstraction to represent resources. As I will better explain in the following sections, they identify also a multi-dimensional space which can be properly used to state and solve a multi-objective optimization problem. From now on, I define:

> *System-Wide Configuration Space (SWCS)* - an N-dimensional space, associated to a specific computing platform, and defined by the collection of SWMs available on the considered platform, represented by the set $M = \{m_i\}$. The dimension N of this space corresponds to the number of SWMs defined, i.e. $N = |M|$. A Cartesian coordinate system can be defined in this space where each axis represents a single SWM, i.e. an element $m_i \in M$.

I use the term "system-wide" because effectively this space allows to represent all the meaningful metrics defined by a system, and thus it allows to represent in a single space all the available "system's resources", with the meaning defined at the beginning of this section.

3.3.3 Devices Abstraction

I already discussed about the structure of modern embedded systems, almost based on SoC, which are composed by multiple different and complex devices, some integrated within a single chip and others external. One of the purposes of an Operating System is just to provide an abstraction of these devices to upper software layers. A device driver manages the complexity of the corresponding device and integrate it within the OS providing a standard interface towards the rest of the system, and to the user-space especially. In instance, within the Linux kernel we can have basically three kinds of device's interfaces: char devices, block devices and network devices. This abstraction allows both applications to transparently access and use devices and at the same time grant portability and interchangeability. Thus: applications do not depend on how a device exactly work, and it is possible to replace a device with another of the same class, without requiring modifications to applications.

The kind of abstraction from bare hardware that device drivers interface provides, and its benefits with respect to user-space interaction, it is a concept that I taken inspiration from when defining a new concept for the proposed solution, more precisely I define:

> *Device Working Region (DWR)* - an abstract representation of a device "working mode", defined by the range of values that it supports for each "sensible" SWM. This representation is defined by the corresponding device's driver and identify a "functional mapping" between device-specific working modes and a region of the SWCS.

A device generally could have different working modes, which correspond to different resources usage and supported QoS. Working modes could be as simple as 'device on' and 'device off', or even more complex such as all the different operating frequencies of a CPU or the different connection protocols supported by a 3G modem. What exactly are the working modes of a device is defined by the corresponding driver. In the technique I propose this corresponds to extend the OS interface that a device driver is required to implement in order to integrate within the system providing a proper hardware abstraction. Actually, it is worth to notice that the implementation of this OS interface extension in not mandatory for the drives, and thus a device could still be used in a system implementing the technique proposed even if it don't define its DWR. Instead, of course the implementation is required if we want the device to take advantages from the optimization framework.
For the convenience of the following discussion, I define $D = \{d_j\}$ to be the set of devices available on a platform, each one with a specific set of supported working mode represented by a set $C_{d_j} = \{c_{jk}\}$. This means that: every device available in the system is represented by an element d_j belonging to the set D, and has a set of available internal configurations represented by the elements of the corresponding set C_{d_j}. Thus c_{jk} is the k^{th} configuration of device d_j. Using $\Delta = |D|$ to denote

the number of devices present in the system, I can represent the set of *system-wide configurations* by:

$$\Gamma \in \{C_{d_1} \times C_{d_2} \times \ldots \times C_{d_\Delta}\} \tag{3.1}$$

that correspond to the combination of the configuration of every single device present in the target system. Of course not all of these system-wide configurations are feasible, in the sense that do not correspond to a usable configuration of the system, e.g. because of hardware dependencies between different subsystems. I will discuss in the following sections how the proposed framework support the identification of the feasible configurations among all these theoretical possible ones.

A device is "sensible" to a SWM when its behaviors are somehow related to the value that this metrics could assume. In instance a CPU with frequency scaling support is sensible to a 'CPU frequency' metric, which model the processor clock frequency as a resource, or a network interface is sensible to the 'Internet Bandwidth' metric, which instead model the network bandwidth as a resource. Among all the metrics that could be defined in a system, each device could be sensible to just a subset of them; in instance the CPU may not be sensible to the 'Internet Bandwidth' metric. Again, as in the case of the working modes, the role to identify sensible metrics for a device, is assigned to the corresponding driver.

Two kinds of relationship between a device and an ASM can be identified:

- a device is *influenced* by a SWM: when its behavior depends on the actual value (or a range of values) of that metric or it is somehow constrained by it. In this case a device d_j depends on the SWM m_i; I formally express this relationship with:

$$influence(p_i, d_j); \tag{3.2}$$

- a device *affect* a SWM: when its configuration, and thus the services it provides, somehow defines the value (or range of values) allowed for that metric. In this case a SWM m_i depends on the device d_j; I formally express this relationship with:

$$affect(d_j, m_i); \tag{3.3}$$

These two kinds of dependency are not mutually exclusive, we can have devices that are both influenced by a SWM and at the same time affect it. If we consider the 'Internet Bandwidth' metric, for instance, this can be either a system requirement for a Wi-Fi network device driver or a system constraint asserted by the device itself based on its actual configuration. In the first hypothesis the device is influenced by the metric, i.e. the device configuration is constrained by the required bandwidth, while in the latter the metric is affected by the device, i.e. the available bandwidth is constrained by the driver configuration. It is also possible to have a SWM depending on a device while the contrary does not happen. In instance, this is the case of a sensor device which state directly affect an hypothetical metric 'Environment Light' while the contrary has not meaning.

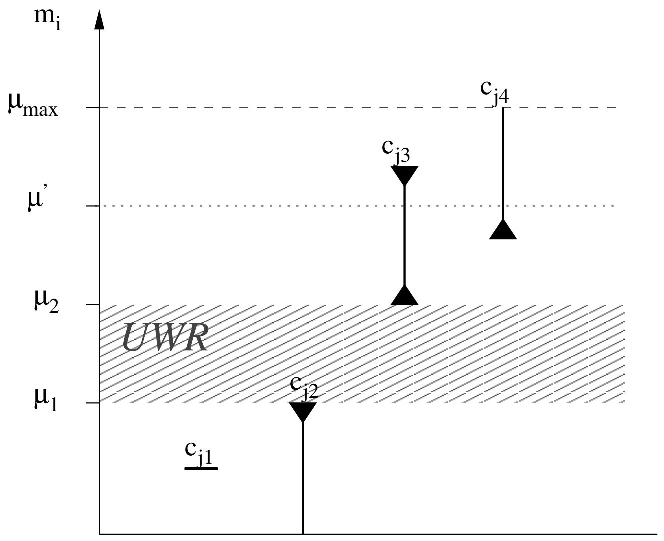


Figure 3.2: Example of working modes of a device d_j mapping on a SWM m_i. Each working mode could maps on a single point (e.g., c_{j1}), an upper/lower bound (e.g., c_{j2}/c_{j4}) or a range (e.g., c_{j3}).

The working modes of a device and its sensible metrics, identified by the driver, are strictly related. More precisely, the sensible SWM are used to formally define the device's working modes:

> *DWR Mapping* - the formal representation of a Device Working Region as a set of constraints, one for each device's sensible SWM. A constraint being a range of values, possibly collapsed on a single value, an upper-bound or a lower-bound defined on a specific SWM.

This mapping could be easily represented within the SWCS, where it identifies for each device's working mode a region, eventually unbounded, of this space.

Graphical representation of mappings

In Fig. 3.2 is represented a convenient graphical representation of the relationships that we can have among the configurations of a device and a SWM m_i, i.e. $\mathit{affect}(d_j, m_i)$. Generally each device configuration can support different values of the metric, e.g. different settings of a network interface corresponds to various bandwidth values available. Therefore we can identify a mapping between each device internal configuration c_{jk} and the metric m_i. More precisely, a configuration could correspond to a *range* of supported values for the parameter (e.g. c_{13}), and eventually just a *single value* (e.g. c_{11}). Ranges can also be single-side limited. In this case, we could have a *lower-bound* when the configuration limits the minimum value of the parameter (e.g. c_{14}), or to the contrary, an *upper-bound* if the configuration limits the maximum value of the parameter (e.g. c_{12}). If we consider a specific value μ' for the metric m_i, a configuration c_{jk} of the device d_j is considered feasible with respect to that value if the value is inside the range defined by the configuration's mapping. If this condition holds we write:

$$satisfy(c_{jk}, \mu') \tag{3.4}$$

It is worth noticing that mappings can overlap, e.g. if we consider the specific value $m_i = \mu'$ in the figure, there are two feasible configurations: c_{j3} and c_{j4}, thus:

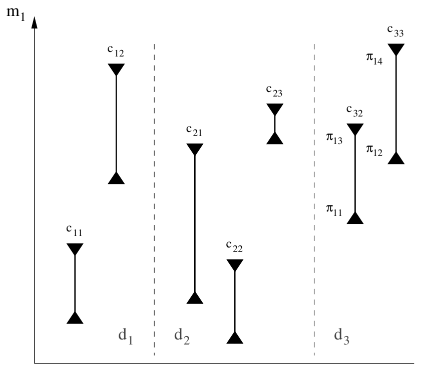
$$satisfy(c_{j3}, \mu') \wedge satisfy(c_{j4}, \mu')$$

Mappings can also define *unfeasible working regions* (UWR) corresponding to ranges of values for a metric which are not mapped by any device configuration, e.g. the range of values defined by $\mu_1 < m_i < \mu_2$ does not have a valid configuration, i.e.:

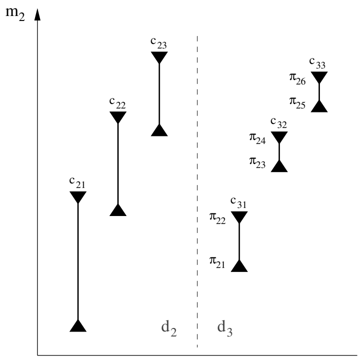
$$UWR: \neg satisfy(c_{jk}, \mu), \forall c_{jk} \in C_j \wedge \forall \mu_1 < \mu < \mu_2 \tag{3.5}$$

I define: $uwr(d_j, m_i)$ as the predicate that defines the UWR of the device d_j with respect to the metric m_i.

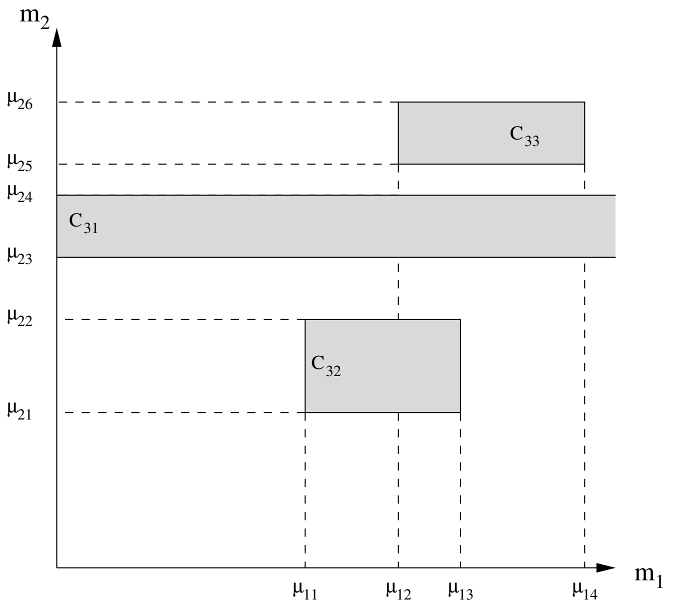
Finally, to show how it is possible to represent DWR within the SWCS lets consider the case depicted in Fig. 3.3 of a simple system with three devices: d_1, d_2 and d_3, and two metrics represented by m_1 and m_2. The mapping of device's configurations with respect to each parameter is supposed to be the one represented in Fig. 3.3a and Fig. 3.3b. It is worth to observe that, in this example, the device d_1 maps only m_1 but has no relation to the other parameter. On the contrary, d_2 and d_3 have configurations that map on both of the metrics. We can also observe that a device can map a parameter only for certain configurations and not for others, such as in the considered example where d_3 does not define a mapping on m_1 for the configuration c_{31}.
It is possible to report mapping informations, represented by the diagrams of Fig. 3.3a and Fig. 3.3b, in a single space representing the SWCS. In this example, it will be a bi-dimensional Cartesian coordinate system whose axes correspond to m_1 and m_2. In Fig. 3.3 are represented the mapping defined by the working regions of the device d_3. As it is shown by the figure it is possible to associate each configuration to a region, possibly unbounded, in the system-wide configuration space. In the considered example device d_3 defines three of these regions, of which one, i.e. that associated to the configuration c_{31}, is unbounded since, as previously noted, in this configuration d_3 does not impose any constraint on the value of the parameter m_1.



(a) Configurations' mapping on m_1.



(b) Configurations' mapping on m_2.



(c) Working regions of device d_3.

Figure 3.3: Mapping between devices' operating modes and system parameters: the basic mechanism that guarantees collaboration among drivers in achieving the goal of control. Devices declare the mapping between their local configuration and the system-wide parameters (ASM) by generating Device Working Regions (DWRs). Figures (a) and (b) consider two ASMs on each axis and show the mapping of devices configurations on such parameters. Figure (c) depicts the resulting DWRs.

3.4 The Model Layer

In this section I define how it is possible to exploit the abstraction defined so fare to build a suitable model of a generic computing system which can be used to efficiently solve the optimization problem of identifying the best trade-off among power consumption and perceived performances.

The behaviors of a real system are defined by the configuration of its many devices. These configurations change time by time, depending on the devices used by user-space applications and the services required to them. Not all devices are always used simultaneously and the resources, either proper of each device or shared among them, could constraint the service level that a device can support.
What I'm going to define now is a model of such a system. This model should take as input the abstract information representing the system resources and capabilities, exploiting the abstraction that I defined in the previous section. The output generated by the model should be an architecture independent representation of all the feasible configurations available for the specific target system. This representation should be sufficiently fine-detailed to keep into consideration important platform specific aspects, e.g. devices inter-dependencies, but at the same time it should be also sufficiently abstract to be used by any optimization policy for the system configuration that we could imagine to develop at an higher level of abstraction.

3.4.1 Tracking devices inter-dependencies

In the application context that I consider, where the system is composed by multiple devices both within the SoC and outside of it, it is possible to have implicit architectural and functional *dependencies* among different devices. This generally imply that a particular configuration enabled on a device could have side effects on another one, for instance when a device is a resource used by another, or when a device affect a metric which is also sensible for other devices, for instance the system latency. More in general, a device configuration can have some sort of influence on overall system behaviors and thus it could affect the services that some other device provides or can expect from the system. In instance, on some embedded platform it happens that slowing down the CPU frequency affects also the external memory bandwidth, and this could have side-effects on others processing engines using the same memory.

Inter-dependencies among different devices could be very difficult to identify and track. When the dependency is defined by architectural design constraints and has a hierarchical structure it is still possible to track them quite easily using ad-hoc frameworks. A classical example of this kind of dependencies is the clock distribution three within a SoC. In this case, the clock framework described in Par. B.2.4 on page 145 is an example of a quite simple framework that can be effectively used to track hierarchical dependencies. Otherwise, if dependencies are not properly tracked, they could lead to suboptimal system configurations or even worst to in-

correct system behaviors.
Moreover, if we consider that devices can be added and removed at run-time, we should consider also that inter-dependencies may be dynamic. Hence, in order to improve driver adaptation to different working conditions and its portability among different systems, it is preferable avoiding to have architecture dependencies embedded directly within the driver's code.

Use SWM to track dependencies. As I showed in the previous section, it is relatively easy to identify how a system's metric is related to the local configuration of a device. Moreover, the usage of SWM is a mechanism that allows implicitly to consider and to track devices inter-dependencies. This mechanism simply require to define how the configurations of a device are related to the system metrics, which it is easier than the effort required to trace all devices inter-dependencies in a centralized way. This approach thus introduce also a layer of abstraction among drivers that both simplifies their development and also makes code more portable among different architectures. Indeed, it is no more necessary to consider how a driver will directly interact with all others in the system, and keep this information updated as long as the system configuration changes over time, but it is required just to statically define how a driver maps its configurations to a set of predefined system metrics. Thus SWMs could be effectively used also as a mechanism to express and track dependencies among devices.

To better explain how the mapping abstraction defined in Par. 3.3.3 on page 70 is suitable to tackle the issue of inter-dependency among device, I use another example considering a simple system with only two devices d_1 and d_2 both having a dependency on the same system metric m_i. The configuration mapping, for instance could be like the one depicted in Fig. 3.4.
Generally two devices could have a different number of configuration states and different mapping between these states and the same system metric. In the example depicted, device d_1 has four configurations, with the mapping of c_{13} and c_{14} partially overlapping. Instead, the device d_2 has only two states that define a discontinuous mapping on the system metric. It is worth noticing that the two devices are not aware of each other. The only information shared somehow is the mapping of their configurations on the metric m_i.
Supposing that the initial system configuration is $\Gamma_1 = [c_{12}, c_{21}]$, then the system is working on a configuration where m_i can have the value μ_1, i.e.:

$$\mathit{satisfy}(c_{12}, \mu_1) \wedge \mathit{satisfy}(c_{21}, \mu_1)$$

Starting from this configuration two different scenarios may happen: either a device would be able to reconfigure itself, as a result of a local optimization policy, or someone in the system requires to support a different value for m_i, e.g. to have $m_i \geq \mu_2$. Both these two events require a system reconfiguration: each device in the system could have to update its settings in order to correctly support the new working conditions.

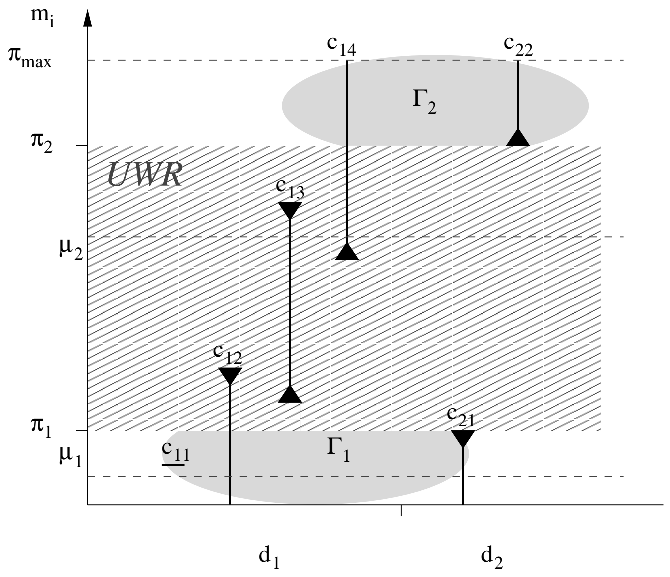


Figure 3.4: Example of two devices, d_1 and d_2, which are sensible to the same system metric m_1. Each device could have a different number of working mode; here d_1 has four states while d_2 has only two. Every state "maps on m_1" defining the range of values for the system metrics that are feasible when the corresponding device is in that state. The unfeasible working region (UWR, ref Eq. 3.5 on page 73) is also represented.

Devices' working mode change. Let us suppose that the optimization policy running locally to d_1 find that there is the need, or simply the opportunity, to change the configuration for instance to better satisfy the system requiring a different service, e.g. we start to playback audio and since the audio-codec device must drive the loudspeakers and mix input channels this affects the system metric m_1 that need to be updated. Let's suppose that the driver identifies c_{13} as a possible new feasible working mode. The mapping shows that this reconfiguration will have an impact on the system property m_i and thus will have a side-effect on the other device d_2. In this case it happens that:

$$range(c_{13}, m_i) \subset uwr(d_2, m_i)$$

that is: the range of the configuration selected by d_1 falls completely within the infeasible working region of d_2, i.e. this last device can not be configured to adapt to the new working conditions. When a situation like this happens, in order to preserve the system in a FSC it is necessary that devices collaborate to find, if possible, a new system configuration Γ where each one is able to satisfy the working conditions. In the example considered the two devices may agree to move the system into the state $\Gamma_2 = [c_{14}, c_{22}]$.

How this agreement can be reached, and what are the mechanisms to support this kind of cooperation among devices, will be better explained in the following sections. What is important to better emphasize here is just that drivers are *loosely coupled*: they are not aware of each other, but their mapping between their local configurations and system metrics is the only information needed to support the identification of system-wide configurations which satisfy all devices.

System metric's change. Different working conditions, e.g. switching to battery power supply, or changing user-space application, e.g. modifying the video quality of a conferencing application, may require an update on some system metric. In this cases, a change on one or more SWM can be adopted to enforce a new configuration to devices that are required to have different behaviors. All the devices that are sensible to an updated metrics may have to update their configuration to remain compliant with the new working conditions. A proper identification and agreement process, to be better defined in the following sections, must ensure that the new system state will still grant a correct system behavior.

In the example reported in Fig. 3.4 we suppose that, while the system configuration is $\Gamma_1 = [c_{12}, c_{21}]$, a lower bound constraint is somehow required for the metric m_i, let say it is required to be $m_i \geq \mu_2$. While driver d_1 could grant the required behaviors reconfiguring its device in c_{13}, as observed before this configuration is not compliant with the unique feasible configuration of d_2 which is c_{22}. Thus both the devices will have to agree about a new system configuration like $\Gamma_2 = [c_{14}, c_{22}]$.

3.4.2 Modeling Feasible Configurations

I justified so far why it is important to properly track dependencies among devices in order to keep the system in a feasible configuration. It is time now to introduce a definition for this concept, thus I define:

> *Feasible System-wide Configuration (FSC)* - a configuration of the whole system where, given a certain value for each system metrics $m_i \in M$, it is possible to find a DWR for every device $d_j \in D$ which is feasible with respect to the given values for the metrics.

Formally:

$$\begin{aligned} \text{FSC} \Leftrightarrow \Gamma = [c_{1,k_1}, c_{2,k_2}, \ldots, c_{\Delta,k_\Delta}] : \\ \textit{influence}(m_i, d_j) \Rightarrow \textit{satisfy}(c_{j,k}, m_i), \forall m_i \in M, j = 1..\Delta \end{aligned} \tag{3.6}$$

According to this definition, when the system is working on a FSC we are granted that each device in the system could be configured to operate in a working mode that don't have any conflict with any other device. Thus, in a FSC any inter-dependency among device is safely solved.

The main goal of the framework I propose is to identify a system-wide configuration which corresponds to the optimal trade-off between power consumptions

and required performances. Thought a number of interesting theoretical techniques could be defined to identify at run-time what is the optimal system configuration, according to both the available resources and the required performance, every outcome is useless if it cannot be actually applied to the real system because of implicit inter-dependencies or hardware constraints ignored by the optimization policy itself. Indeed an optimized configuration cannot be identified regardless of its feasibility. Thanks to their interesting property, the identification of all system's FSC is especially important for the definition of such a framework.
In the light of these considerations, the optimization technique that I propose is based on the *a-priori identification* of all and only the system feasible configurations. Thus, any optimization policy that will be developed on top of this framework, it will be granted to operate on a set of real and valid configurations and consequently each result can be safely applied to real system.

FSC identification and the proposed approach. As described in the overview, where in Sec. 1.5 on page 19 is presented the fundamental approach of this thesis, among the three main steps of the proposed technique the *FSC Identification* is the really first one. This step allows to satisfy both the 'system-wide' and 'fine-detail' design requirements of modern systems. Indeed FSCs represent system-wide configurations which take into considerations all the devices along with their resources concurrently and, how I will discuss in the following, also the requirements coming from all applications as well. Besides, the definition of FSC itself, which is based on the abstractions of SWMs and DWRs, allows to consider platform-specific fine-details such as device's specific operating modes and inter-dependencies related to architecture design constraints.

The identification of FSC is one of the main outcome of the model that I defined on top of the underlying abstraction layer. This model use as input the DWRs defined by device drivers and the SWCS defined by the set of SWM available in the target platform. The output provided by this model instead is represented by the set of all the FSC which can be identified. This model thus provides a description of the feasible configuration of a real system according to the available resources, the working mode of the devices sensible to these resources and all the inter-dependencies that could occur among the devices. I will details how the FSC can be used to solve the optimization problem in the following sections, in the rest of this section instead it is worth to explain how all the FSCs can be identified, and for simplicity I will do this using a graphical example.

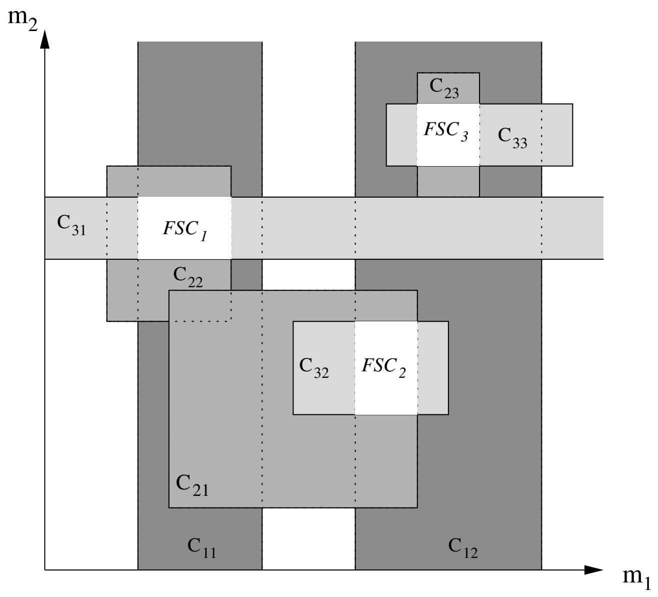


Figure 3.5: The FSC defined by the mapping of three devices on two system metrics m_1 and m_2. Each c_{di} area represents the *i–th* working region (DWR) of the *d–th* device. If all DWRs are reported on the same graph, the existing overlap can be highlighted. The overlapping of DWRs identifies a subspaces of the SWCS that defines acceptability ranges for system-wide parameters for all the corresponding devices.

Graphical identification of FSC. The FSC identification procedure can be efficiently implemented with an imperative algorithm. For the detail of a possible implementation I forward the reader to Par. A.5.2 on page 136. In this paragraph instead, I will give an intuitive explanation of the identification procedure using a graphical approach.

Lets consider the simple system initially described at page 72. The example considered only three devices and two metrics. In Fig. 3.3a and 3.3b was represented the mapping defined by each device. Instead, in Fig. 3.3c, for clarity of exposition, was represented only the DWRs for d_3. If in this last figure I add the DWRs defined by the device d_1 and d_2 along with the ones already present for d_3, we get a representation similar to the one depicted in Fig. 3.5. From now one I will refer to this last figure.

First of all, it is worth noticing that generally we can obtain *overlaps* between different DWRs, either between those of different devices or also referring to the same device, e.g. as it is in the case of c_{21} and c_{22}. Generally, the overlap of two DWRs identifies a subspaces of the SWCS which defines acceptability ranges for the system metrics. Thus, these subspaces correspond to a restriction of the acceptability ranges with respect to those defined by the individual overlapping DWR. I define:

> *Overlapping DWRs (ODWR)* - a regions of the SWCS in which they overlap two or more DWRs corresponding to different drivers.

Therefore, on the base of this definition we must observe that, in the example depicted in Fig. 3.5, the region corresponding to the intersection between c_{21} and c_{22} is not an ODWR, since this two DWR belongs to the same device d_2. But, if for the same region, we consider also the overlap with the DWR c_{11} defined by d_1 than this could be considered an ODWR.

Deeper analyzing ODWR, we can notice that there are particular regions of the SWCS where there exists the overlap for at least a DWR for each device. In Fig. 3.5 these regions have been represented with a white background. These regions are particularly important since they identify ranges of system metrics that always correspond to a valid configuration for each device; therefore they actually correspond to the FSC as previously defined. Thus, I can give the:

> *Geometric definition of FSC* - a FSC is geometrically represented in the SWCS Cartesian space by an ODWR defined by the overlapping of *at least one DWR for each different device* present in the system. These regions are convex by definition.

In the simple example of Fig. 3.5 we can easily identify three FSC corresponding to these combinations of devices configurations: FSC$_1$ $[c_{11}, c_{22}, c_{31}]$, FSC$_2$ $[c_{12}, c_{21}, c_{32}]$, and FSC$_3$ $[c_{12}, c_{23}, c_{32}]$.

3.5 The Optimization Layer

In this section I define how it is possible to exploit the model defined so far to support both the definition and the usage of a set of policies to efficiently solve our optimization problem, i.e. identifying the best trade-off among power consumptions and perceived performances.

A real system, especially those designed for multimedia mobile applications, are subject to frequently changing working context. The usage of different applications, the changes in resources availability (e.g. battery or wall powered) or environmental operating conditions (e.g. different network connections availability), usually require a system reconfiguration to keep in pace with new usage requirements. A system reconfiguration could be triggered also by a change in the optimization strategy. For instance, passing from a power aware to a performance boost strategy, in response to an increased power budget (e.g. the device has been connected to the wall plug), usually change the optimization target and thus the system configuration has to be updated accordingly.

The optimization technique that I propose exploit the system view offered by the FSC model and define a suitable strategy to assign a "weight" at each feasible configuration, according to the running optimization policy. How this weight could be defined is the subject of this section, the basic idea anyway is to be a sufficiently abstract metrics to easily adapt to a generic *multi-objective optimization*. The *run-time tracking of application requirements* has been another goal of the optimization policy I defined. One more time, the abstract system model based on the concept of FSC and their representation in the SWCS defined so far, has been properly exploited to translate application requirements on constraints for the problem of searching the optimal configuration. Indeed, in this section I will show how it is possible to formulate the optimization problem as a constrained optimization problem and then how to solve it using an empirical but efficient approach.

Before to digg into the optimization layer details, it is worth to start with an overview of the proposed control solution, which at the end it is configured as a classical hierarchical distributed control system.

3.5.1 Hierarchical distributed control

This work advances the proposal for a control and optimization approach that improves both implementation simplicity and portability without reducing too much the control accuracy. Highly efficient and precise centralized controls solutions, such as DPM [97] and many others among those reviewed in Sec. 1.4 on page 16), exhibits some limitations mainly related to their implementation complexity. To overcame these limitations the technique I propose use a "divide and conquer" approach by splitting the system-wide control problem into two different sub-problems: low-level devices local controls and an higher-level distributed agreement control. An overall view of the proposed solution is depicted in Fig. 3.6.

In this model a driver can exploits the fine-detail knowledge on the capabilities of the controlled devices to run a *local optimization policy*. This policy allows to fine-tune the devices' configuration based on the system requirements and working conditions. It is worth to notice that drivers are usually autonomous entities developed independently from other system components. Nevertheless they should be aware on the possible side effects that each local configuration could have on other system components. The mode I propose simplify the implementation of such an awareness with the introduction of the quite simple and sufficiently abstract concept of SWM. These metrics not only allows to decouple the local policies among them but also they introduce an abstraction level between local policies and the higher optimization layers. Indeed, thanks to these metrics, drivers can indirectly share informations by simply defining: a) how each local configurations could affect these parameters and b) when an update on these should influence the device configuration.

This exchange of informations allows not only the enforcing of system requirements down to local policies, but also the "collaboration" among devices in order to move the system towards an optimized working configurations. This cooperation among local policies is transparently achieved with a *distributed agreement process*,

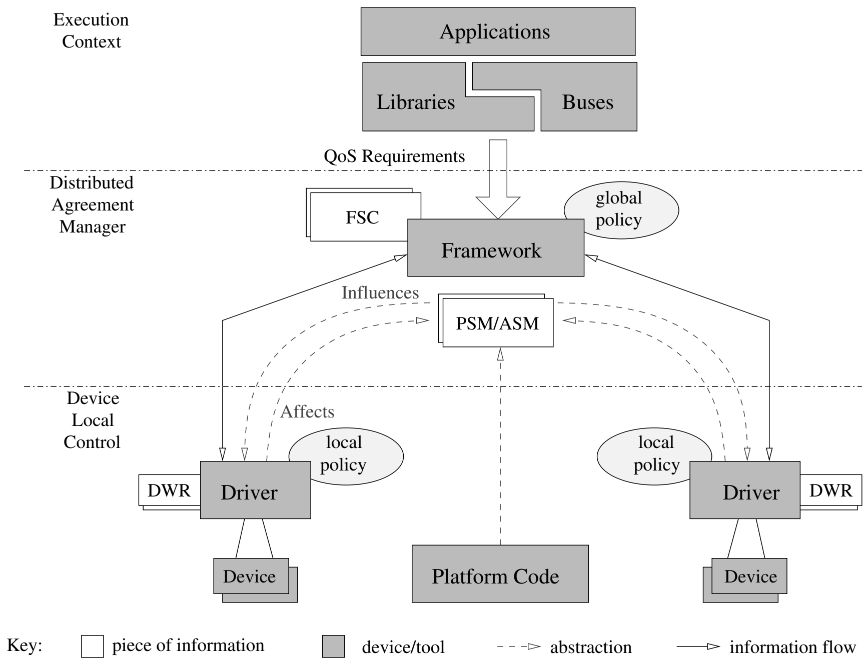


Figure 3.6: Hierarchical distributed control. The proposed solution splits the control problem in two sub-problems: drivers' local control and global agreement manager. The drivers exploit fine-detail knowledge of controlled devices to fine tune them according to a local policy. The management of devices inter-dependencies and QoS requirements instead is handled by a distributed agreement manager. This is implemented within an higher-abstraction level framework which exploit a model of the underlying system based on the description provided by FSCs.

which is the highest optimization level of the proposed hierarchical control. The global optimization policy implemented by this higher abstraction level exploits both low-level informations, related to resource availability and hardware capabilities, and also high-level informations, related to applications' QoS requirements.

All that reasoning motivate the classification of the proposed techniques as *hierarchical distributed control* system, with a global system-wide optimization policy in the upper layer and many local fine-tuning optimization policies in the lower layers. The space and time allowed by a doctoral thesis permit the complete analysis of just one of these many layers. I choose to focus my attention on the upper layer and thus in the rest of this section I discuss the definition of a global optimization policy. This choice has a double motivation, on one hand it is the more interesting part, on the other many local optimization policies have already been investigated. The upper layer thus is the more interesting, also because, being the more abstract layer, the designed solution will be completely platform independent and thus it can be directly implemented within the framework I developed. Instead, lower layer's policies must be strictly related to the devices and thus they require a detailed analysis of each specific device class, which could itself require a complete thesis work. Moreover, as I documented in the prior-art Sec. 1.5 on page 19, many researches have already focused on the definition of local optimization policies for different classes of peripherals. All these theoretical contribution could be easily integrated within the proposed framework, just when a proper global optimization strategy has been defined.

3.5.2 A theoretical approach to the optimization

The distributed control optimization problem described so far, can be conveniently reformulated using an appropriate formal model. A transposition of this type not only provides a rigorous description of the problem and a formal proof of the solutions quality, but also allows to more easily identify possible alternative ways for its solution by exploiting the particularities of the formulation instance. To this purpose, I decided to use *Linear Programming*. This choice has two main reasons: from one hand, the problem formulation that I'm going to describe was foreseen to easily fit within an LP model, and from the other, LP is a well known and adopted optimization framework.

Before proceeding with the description of how we could move towards an LP formulation of the optimization strategy, it is worth to notice and stress a point. Even if it could be possible to effectively solve an LP problem to identify the optimal system configuration, I don't want to adopt that approach in the framework I propose. I am aware that it exist a number of libraries providing highly optimized algorithms for the solution of different flavors of LP problems. However, my target is just to show the it exist an established formalism for the verification of existence of a solution and eventually its identification. Indeed, if I'm able to design a different strategy which can be proved getting to the same results, than I'm granted that also the solutions identified by this last strategy are optimal, without any additional burden of proving. This is an important point to stress, because the actual implementation of the optimization framework I proposed is intended to be integrated within the Linux kernel where, for reasons of efficiency and codebase generality, it is not present any LP solving library and much probably it will never be integrated. A second reason, but still not less important, for this my choice is the foreseen possibility to provide an hardware acceleration support to the optimization technique. In the light of these considerations: for the effective design of the solution strategy, which is the real subject of the following subsection, I choose to stick to the "design-for-changes" paradigm and thus the implementation flexibility will be a key target for the actual definition of the solution strategy.

The graphical representation of FSC described on Par. 3.4.2 on page 80 is particularly convenient to state our optimization problem in terms of LP. A problem of linear programming requires the identification of an optimal solution, given: a solution space, a set of constraints on it and an objective function. Let me review in details each one of these elements.

The solution space representation

The optimization space of the LP problem is represented by the SWCS. This is a quite intuitive translation if we consider also the definition of SWM, given on page 67, and that I already observed that these metrics identify also a multi-dimensional space which can be properly used to state and solve a multi-objective optimization problem. Within this space an LP formulation require to identify a particular region which correspond to the "valid solutions locus", i.e. the region of point which are the feasible solutions of the problem among which is the optimal solution. Let's show how such a region can be represented within our model.

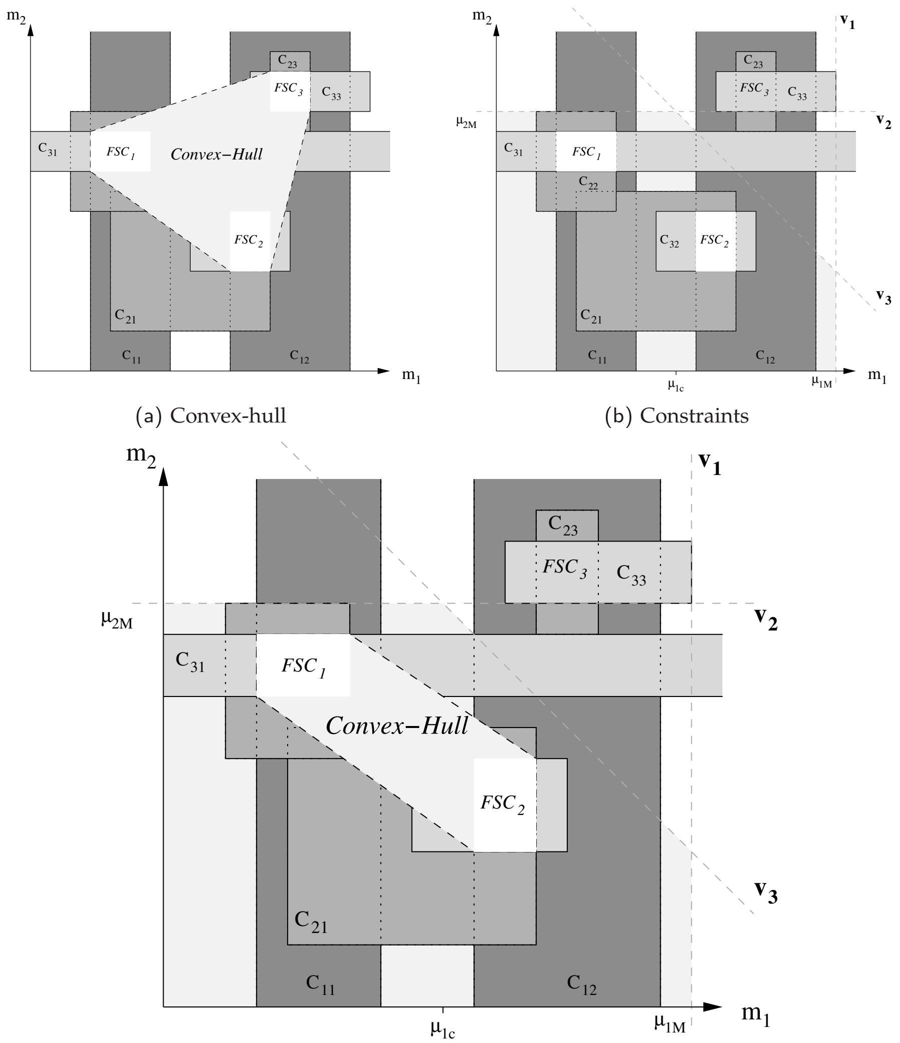
We already know that the solution of our optimization problem is represented by a point in the SWCS. We know that the regions corresponding to FSCs identify the only valid combinations of device's configurations of the system. Therefore, the solution of our optimization problem must be a point in the SWCS that belongs to one of its sub-regions defined by the FSCs. It is worth noticing that:

> Def. 3.1: Every point within a single FSC's region identifies a *unique solution* for our optimization problem. Indeed, each region represents a well defined combination of device's configurations.

For instance, in Fig. 3.5 on page 80 every point belonging to FSC$_1$ always corresponds to the solution: $[c_{11}, c_{22}, c_{31}]$.
This means that every point of a FSC's region is equivalent in terms of the problem solution. We can then assume that the optimal solution of our problem always belongs to points that define the border of a FSC's region. This assumption is particularly interesting if we consider that the optimal solution of a well defined LP problem is always on the border of the solution space. From these considerations, it is easy to be convinced ourself that:

> Def. 3.2: The *Solution Space (SS)* of our problem is defined by the smallest convex polygon that contains all the *valid* FSC's. This region is known in LP as *convex hull*.

Thus, in the considered example the convex-hull is depicted in Fig. 3.7a.



(a) Convex-hull

(b) Constraints

(c) Shrunk Convex-hull

Figure 3.7: Linear Programming formalization. a) FSCs are the only regions of the solutions' spaces that include system-wide configurations valid for each device. Every point within a single FSC region identifies an unique solution. b) Due to system evolution, additional constraints on SWMs can be asynchronously asserted by devices. They are generally represented by a surface in the N-dimensions space. In the above example, where a two dimensions space is considered, they correspond to a simple line. c) Constraints shrink the solution space since some FSCs becomes invalidated.

The constraints representation

In a generic LP problem the convex-hull's border is defined by the constraints imposed for the resolution. In our particular situation such constraints are not explicitly given but instead are implicitly defined by the identification of the FSCs. Indeed, starting from the FSC we can go up to the equations of the constraints that define the convex-hull. Nevertheless this is not a limit but just a different way of formulating the constraints of the problem to be solved. Seen from another perspective, this means that knowing FSCs is fully equivalent to know the constraints that define the convex-hull within which to seek the solution. For that reason, I define:

> Def. 3.3: *Implicit Constraints (IC)* - the set of constraint on the SWCS, which are deducible by the knowledge of FSCs, that identify a convex-hull corresponding to the smallest convex polygon containing all the FSCs.

Discussing about the application context and its nature of being a discrete event system, on Par. 3.1 on page 62, I observed it may happen, at some point in time, that it is necessary or perhaps just possible to change the system configuration. Once that happens, the identification of a new FSC will have to take into consideration all the requirements on the system metrics which could be asserted by either applications and drivers. These requirements, in the case of problem's formulation using LP, are just like additional constraints on the solution space. To distinguish these constraints form the previous ones, I define:

> Def. 3.4: *Explicit Constraints (EC)* - the set of constraints on the SWCS, which represent the requirements, that are expressed by both user-space applications and drivers, on the values of SWMs.

An example of explicit constraints is shown in Fig. 3.7b, where three constraints v_1, v_2 and v_3 have been introduced to require that a valid system configuration satisfies these inequalities:

$$(p_1 \leq \pi_{1M}) \;\wedge\; (p_2 \leq \pi_{2M}) \;\wedge\; ((p_1 + p_2) \leq (\pi_{2M} + \pi_{1c}))$$

Considering constraints of this type does not invalidate the definition of solution space, previously stated on page 85. But a better definition of what I mean by "valid FSC" is still required:

> Def. 3.5: *Valid FSC (VFSC)* - a FSC, or a restriction of it, which is compatible with the current requirements on SWM, i.e. the constraints on the SWCS.

It is worth noticing that, according to this definition, an FSC is valid even if a constraint partially cut it. In this case the what is valid is its restriction, which is geometrically represented by the subtraction from the FSC region of the space invalidated by the constraint, and considering the remaining space as the new FSC. A formal definition for that is:

> Def. 3.6: *Restricted FSC (RFSC)* - with respect of a constraint v_i, is the new FSC that we obtain by the intersection of the original one with the validity space defined by the constraint v_i.

It is worth to notice that, according to this definition, the restriction with respect of a constraint, of an FSC which is not cut by that constraint, is equivalent to the original FSC. This means that during the identification of the convex-hull, for the formulation of the LP problem, it is simply enough to consider only the restriction of all FSC with respect to any current constraint.

For instance, in the situation depicted Fig. 3.7b, the FSC_3 appears no longer valid for the definition of convex-hull and thus it should not be considered. The new convex-hull to solve the LP problem is represented in Fig. 3.7c and is built considering only the regions associated to FSC_1 and FSC_2, that are still valid with respect to explicit constraints.

Objective function identification

The last element to define to complete the formulation of our optimization problem in term of LP is the identification of an objective function. In linear programming the objective function is defined as a vector in the solution space that identifies the optimization direction. The optimization direction lets us explore the solution space in order to identify the optimal working points. It is worth to notice that this corresponds at performing a multi-objective optimization. Indeed, in the exploration of the solution space for the research of the optimal configuration, we consider simultaneously multiple directions, each one representing a goal.

By their definition, system-wide metrics represent resources and thus are strictly related to QoS levels. For instance, we may consider that m_1 is the 'CPU latency' metrics, representing the maximum time allowed to respond an interrupt, while m_2 is the 'Bus bandwidth' metrics representing the bandwidth required in the system bus. Optimize the system, in this specific example, may require to find a configuration which best satisfy this system-wide optimization goal: reduced CPU latency consistently with having the highest bandwidth available on the system bus. These abstract requirements define each one an optimization goals for each individual metrics, and thus in turn they simply define a direction in the SWCS that can be represented by a vector. These vectors can be mutually composed in order to identify the overall system optimization direction, and thus representing the aforementioned system-wide optimization goal. Moreover, we can also imagine that we want to maximize the performance of our system using different efforts on each optimization direction. In this case, the simple usage of a "weight" associated to each single SWM allows to fine tune the optimization direction, giving greater importance to certain metrics than others.

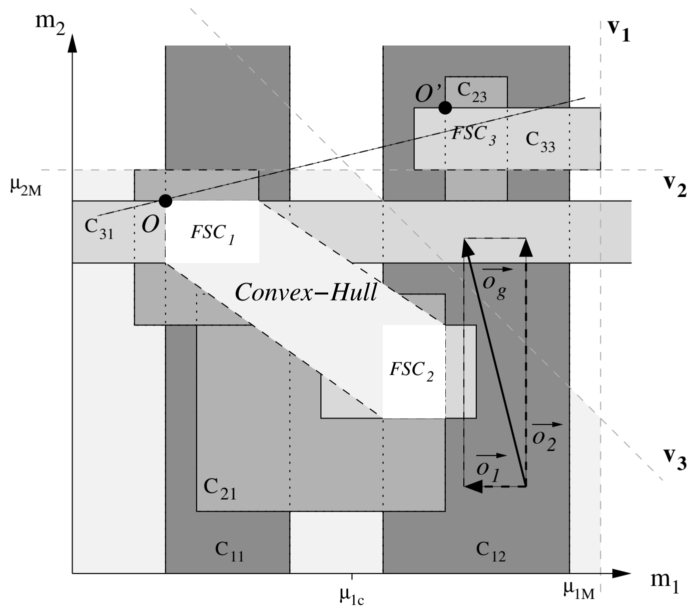


Figure 3.8: Objective function and optimal solution. Objective function is represented through a vector, opportunely oriented to indicate the direction of the optimal solution which depends on the QoS metrics to be optimized.

These concepts are represented graphically in Fig. 3.8 where I depicted both vectors $\overrightarrow{o_1}$ and $\overrightarrow{o_2}$, corresponding to the optimization objective of each individual metrics of the example, respectively: m_1 and m_2. In the same figure $\overrightarrow{o_g}$, which is the vector obtained by the composition of the previous two, represent the global objective function. This last vector identifies an optimization direction, and thus I define:

> Def. 3.7: *Improvements Direction Vector (IDV)* - the vector $\overrightarrow{o_g}$, representing the optimization requirements, which define a direction in the SWCS that correspond to improving solutions, with respect to these optimization requirements.

The solution of the LP problem is obtained from the convex-hull, being defined by implicit and explicit constraints, considering the direction defined by the objective function. The points of the convex-hull that are farthest in the direction indicated by the optimization vector $\overrightarrow{o_g}$ are the solution of LP. In the same example of Fig. 3.8 the direction of $\overrightarrow{o_g}$ indicate that O is the solution to the LP problem. It is worth noticing that if the explicit constraints change so that FSC_3 is not excluded in the construction of the convex-hull, then the optimal solution, considering the same objective function, would becomes O'

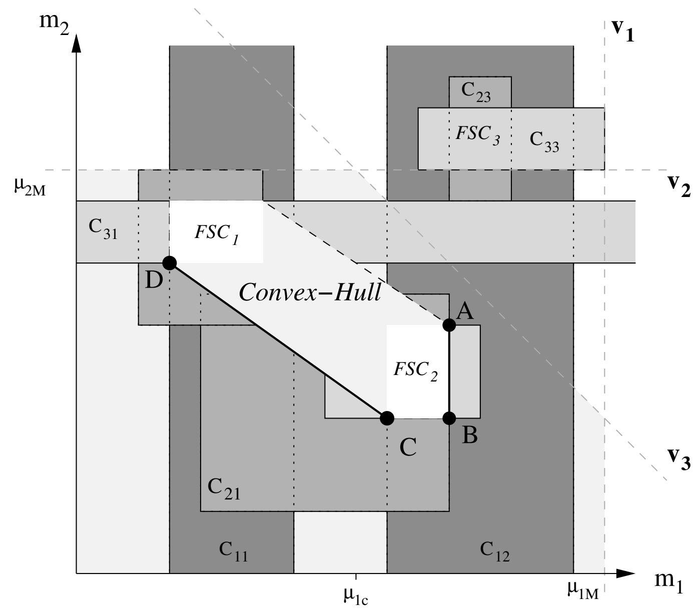


Figure 3.9: Mapping of LP solution back to the original problem. The solution of an LP problem is always located on the border of the convex-hull, and can be a vertex (e.g., A) or a segment (e.g., $\overline{AB}$). Whatever the solution is, it identify always at least one FSC, which are indifferently the optimal solutions of the original system-wide configuration problem.

From LP solution to the optimal configuration

The solution to a LP problem is proved to be always on the border of the convex-hull, and it corresponds to a vertex if we have a single solution or to an entire segment of the polygon in case of multiple solutions. However, our specific problem requires the identification of one or more FSC that are optimal with respect to active constraints. Therefore, it is required to understand how translate the solution of the LP problem, which I described so far how can be obtained, on a FSC among those defining the convex-hull.

It is easy to convince ourself that every vertex of the convex-hull belongs to exactly one and only one FSC, this come from the definition of convex-hull. Thus, if the solution to the LP problem corresponds to a single convex-hull's vertex, the solution of our problem simply corresponds to the FSC which the vertex belongs to. For example, if we consider that the solution of the LP problem depicted in Fig. 3.9 is the single vertex A then this means that the only feasible configuration for our system is the one corresponding to the FSC_2.

Instead, when the solution of the LP problem corresponds to a segment of the convex-hull we have two possibilities. If this segment belongs entirely to a single

FSC region, then we fall back in the previous case, and this configuration is the optimal one. Otherwise, the solution's segment will combine two different FSCs and this means that both configurations are optimal. Indeed, in this case the solution of the LP problem states that all points of the segment are equivalent in terms of objective function, and hence the two FSCs turn to be equivalent.
This two cases are depicted in Fig. 3.9: if the solution is the segment $\overline{AB}$, then we still have that FSC_2 is the only optimal system configuration, otherwise, if for instance the solution of the LP problem turns to be the segment $\overline{BC}$, then both FSC_1 and FSC_2 are equivalent optimal configurations.

3.5.3 An empirical approach to the optimization

The word empirical denotes information gained by means of observation, experience, or experiment[2]. A central concept in science and the scientific method is that all evidence must be empirical, or empirically based, that is, dependent on evidence or consequences that are observable by the senses. This is exactly the approach I adopted: starting from the observation of an hypothetical optimization strategy, implemented according to the theoretical approach previously described, I want to derive an implementation which is empirical equivalent to it. The reason for such an approach, as I discussed in Par. 3.5.2 on page 84, is mainly related to the flexibility of the implementation.

In the overview chapter, talking about the fundamental approach of this thesis and with the support of the Fig. 1.5 on page 20, I described how, starting from the focused application context and the corresponding requirements for the design of modern embedded systems, it is possible to derive an optimization technique which is fundamentally based on three steps: FSC identification, ordering and selection. Now I have all the elements to better explain this claim.
The target of the optimization layer is to identify at run-time what is the best feasible system configuration. I already observed that all and the only possible solutions to the optimization problem are represented by the FSCs, which define an abstract representation of the target system and are identified by the model layer. Therefore, the target of the optimization layer could be refined into: *identify at run-time what is the best FSC among those offered by the model layer.*

The identification of all the FSCs provided by the model layer is the first step of the proposed optimization technique. How much a certain FSC is a good solution depends on the optimization objectives, and thus on the running global optimization policy. This represents the second step of the proposed optimization technique. And finally, whether a certain FSC is feasible or not at run-time depends on the set of active constraints, that is whether it is valid or not according to the definition given on page 87. Thus this is exactly the third step of the proposed technique. Let's review now the details of each one of these steps.

[2]Definition from "The American Heritage Dictionary of the English Language, Fourth Edition"

I STEP – Platform-specific FSC identification

According to the proposed technique described so far, the identification of FSC is in charge to the model abstraction layer. However I prefer to detail the approach implementation in this section, not only for presentation consistency with the next two steps of the proposed technique, bu also because it is itself subject many different optimizations. Indeed, even if it is possible to identify an initial simple but still effective implementation for this step, as I will motivate and discuss later, it is also possible to further optimize this step thus improving the overall performances of the complete solution.

The *FSC identification* is the first step of the optimization working flow, and it is necessary to build an abstract model of the real system which define a knowledge base that the upper layer could exploit to manage and supervise the actual platform optimization. The main goal of this step is to acknowledge data coming from devices and then generate the set of FSCs on the base of these data. A driver, usually when it registers itself within the OS, defines the DWR of the controlled devices and publics this information. The model layer collect these informations from all the registered devices and exploit them to synthesize the FSCs which will then be available to the upper-layer for the optimization of the platform.

A simple algorithm for the FSC identification is based on the depth-first search on a tree-like data structure. I forward the reader to the relevant appendix describing the proposed implementation for the full details on how this is done. What is important to comment here instead is the overhead of this step. Indeed, one of the requirements for the design of new generation control solutions is to have very low-overhead at run-time.
The complexity of the algorithm proposed, which is actually analyzed in the following chapter, is well known to be exponential. Of course, one may complain that this corresponds to considerable drawbacks on both the solution scalability and even more important on the energy overhead introduced by this computation. *Does we risk to spend more power to prepare the ground for the optimization than the power which could be saved by the following optimizations?* This is a licit question and thus by counter some observations are worth to be done.

The complexity and the consequent time and energy overhead introduced by the FSC identification, is mitigated by the fact that this operation is just *seldom performed*. It is actually required only when a new device register itself int the OS. This happens usually at system boot time, when most of the available devices are scanned by drivers and properly initialized. Otherwise, the identification is required when a new device is hot-plugged into the system. Anyway, even on desktop and laptops, these operations are usually quite infrequent and it will be demonstrated that they have a reasonable time frequency which is order of magnitude bigger than the time required by the identification processing. This means that the energy required to build the model is foreseen to be well compensated by the following optimization advantages.

Optimization opportunities Moreover, if we specifically consider an embedded system like a smart-phone or a GPS navigation system, it is reasonable to think that all devices are present from the beginning and in most of them it is even impossible to hot-plug new ones at runtime. Considering this specific scenario, a rich set of optimizations can thus be easily identified to mitigate even more the complexity of identification stage.

Boot-time identification – The identification can be performed just when the system boots. The set of FSCs identified can than be available for the entire duration of the working session of the device, until a new reboot is done, since we are granted that no other devices can be added or remove from the system.

One can complain that this approach increases boot time, which is a metric that nowadays influences QoS and the user experience. Indeed one of the main topic of current industrial research in the optimization of modern Operating Systems like Linux, especially when used in embedded multimedia mobile environment, is the reduction of the delay between the power-on and the moment when the user interface is available with a device which is fully functional.

Considered this specific aspect, it is worth to notice in advance that even the time required for the identification of quite complex scenarios, according to the experiments presented in the following chapter, is at least one order of magnitude slower than the most aggressive nowadays boot-up requirements. Anyway, a simple and effective solution to this problem is to run the identification process only when the user interface has been already loaded. The FSCs identification can run in background, without compromising system response, and take control on the system configuration only once a system model has been properly identified.

Off-line identification – In the specific case of embedded system without hot-plug support the complete set of FSCs could be generate just one time, eventually off-line when a new product is designed and tuned. This set of stable FSCs can than be stored in persistent memory and simply loaded at boot-time. In this way the overhead introduced by the identification algorithm is actually equal to zero.

Hardware assisted identification – Another interesting opportunity, which is foreseen as a future extension of this work, consists in implementing the algorithm with a dedicated hardware companion chip that could accelerates not only the identification of FSCs but also provide support for the other two steps of the proposed technique. This possible extension will be better discussed in the relative section of the concluding chapter. Such a solution could be especially interesting for emerging SoC based embedded devices with a wide number of devices and high need of very aggressive and low-overhead optimization.

II STEP – Policy-based FSC ordering

This is the first step which is properly related to the run-time system optimization. In the previous theoretical approach I discussed how LP can be used as a suitable mechanism to identify the optimal solution. If we look at how works some of the main algorithms for the efficient solution of linear programming problems [98] we could get some inputs on how the solution is identified. Basically the idea is that: starting from a vertex of the convex-hull which identify a generic feasible solution, we move along the border of the convex-hull in the direction that better improve the solution. This step is repeated until the solution cannot be further improved. Thus the optimal solution could be found by successive refinements, thanks to the convexity of the explored border. What I propose for the actual implementation of the technique is something similar to that reasoning, but with a simple variation: the successive refinements are pre-computed once the optimization objectives have been defined for each FSC.

More precisely, the proposed technique computes the goodness level of each FSC according to the optimization objectives. This is a quite simple result to achieve if, according to the discussion on how to use the LP solution given on page 90, we remember that:

a) the optimal LP solution is always found on a vertex of a RFSC

b) every point within a FSC corresponds to the same system configuration

Indeed, to describe the goodness of the system configuration represented by a FSC, we have simply to associate to each FSC a unique goodness level which is equal to that of one of its vertex. This value is than used to generate a list that define a partial-ordering of all the feasible configurations. Thus I define:

> *Ordered FSC list (OFL)* - a list defining a partial-ordering of all the FSC, with respect of their goodness level defined by a certain optimization policy.

The OFL can be exploited by a mechanisms which represent something similar to the successive refinements algorithm previously described. Indeed, starting from a generic element of the OFL, and moving towards the proper direction we always get to an improved system configuration. This list as another important property: it allows to compare two RFSC and find which is better between them, with respect to the optimization policy used to build the list, by simply comparing their position within the list.

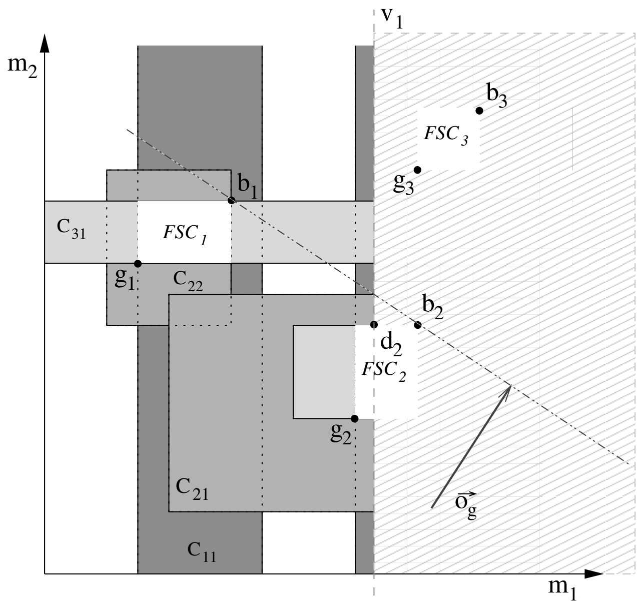


Figure 3.10: The goodness levels of a FSC. Given a FSC (e.g., FSC_2) and an objective function (e.g., o_g), each feasible configuration define two static goodness levels: a best-effort and a granted level (e.g., b_2 and g_2). Moreover, once a constraint is asserted at run-time (e.g., v_1) the actual goodness level of a restricted FSC could change, this is the dynamic goodness (e.g., d_2).

Best-effort vs Granted QoS ordering. So far, I described how it is possible to build a list of pre-ordered RFSC, by simply associating to them a goodness level which is equal to that of one of its vertex. The problem now is to define how to choose this vertex. The approach based on the LP comes to our aid once again. Every RFSC, by its definition, is represented by convex region in the SWCS. Thus, considering a generic objective function: the corresponding direction in the SWCS identify exactly two points for every RFSC. Lets have an example, considering the simple scenario depicted in Fig. 3.10, where in a 2D SWCS we have only two FSC. The nearest point in the optimization direction, named g_i, belongs to the set of vertexes G, and it identifies a *granted QoS configuration*. For example, if in the figure we consider FSC_2 and the corresponding system configuration, we are granted that this configuration will ensure the QoS level represented by the vertex g_2. Indeed, any single configuration point within FSC_2 will have for sure a goodness level greater than that of is representing vertex. Reversing this reasoning, the point which is farthest in the optimal direction, named b_i, belongs to the set of vertexes B, and it is easy to convince ourself that it identifies a *best-effort QoS configuration*.

In the light of these considerations, we understand that in the definition of the optimization policy, along with the weight that can be associated to each SWM, to define the orientation of the objective function's vector, it is possible to define a further attribute. This attribute could express how to choose the representing vertex of each FSC. Not only it is possible to choose between a best-effort or a granted approach, but potentially it is also possible to configure alternative strategies, e.g. a weighted average between this two bounds.

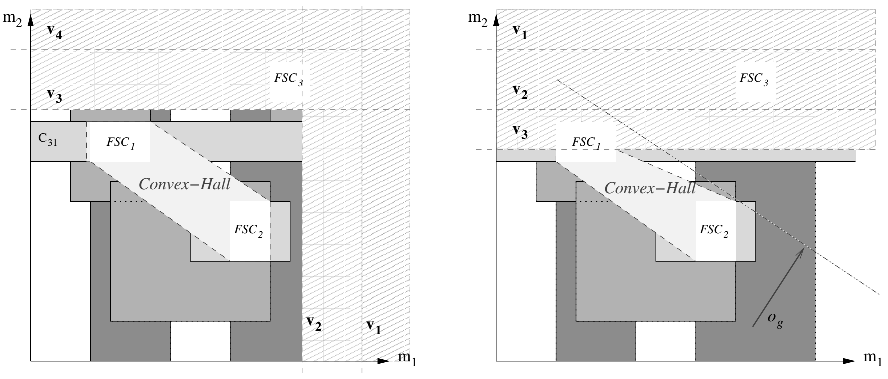
Dynamic goodness. A further observation is worth to be done regarding the FSC ordering while considering run-time constraints. It could happen that at run-time, some application or driver requirement define a constraint in the SWCS which partially cut a FSC. I defined RFSC the restriction of a feasible configuration defined by constraints. What is important to consider is that a restriction could impact on the FSC ordering. Indeed, the goodness level of a RFSC can be different from that of the original FSC. This difference depends also on the method adopted for the choice of the representing vertex. To properly handle these situations, I define:

> Def. 3.8: *Dynamic goodness* - the QoS level associated to a RFSC; it can never be greater than the corresponding *static goodness* which is the goodness level of the original FSC.

For example, lets consider still the simple scenario depicted in Fig. 3.10, and RFSC_{2,v_1}, which is the restriction of FSC_2 with respect to the constraint v_1. In the case of granted QoS ordering the dynamic goodness is not affected and thus the ordering is not changed by the consideration of v_1. Vice versa, in the case of best-effort QoS ordering the dynamic goodness of RFSC_{2,v_1} no only is lower than that of the original FSC_2 but it is also lower than the static goodness of FSC_1. In this case thus the ordering of FSCs is affected by the assertion of the constraint v_1.

In order to be empirical equivalent to the LP theoretical approach, the dynamic goodness effect should be considered by the implementation of the proposed technique. However, I forward the reader to the section describing the implementation details, where it is explained how this effect could be efficiently handled without compromising the effectiveness of the a-priori FSC ordering based on the static goodness.

Finally it is worth to notice that this step is required only when the optimization policy is changed. Indeed, a change to the optimization goals, implies a modification of the objective function and of the corresponding optimization direction in the SWCS. Thus the static goodness of each FSC generally change and this could lead to a different static ordering of the feasible configuration. Beside being the optimization policy change a relatively infrequently event, the complexity of the algorithm for the FSC ordering is essentially linear in the number of FSC. Indeed, the time and energy overhead corresponding to this operation could be easily compensated by the benefits that the pre-ordering introduce at run-time as I explain discussing the following step. Moreover, the hardware extensions sketched in the description of the previous step, are foreseen to be effectively used also to speed-up this step. Once again



(a) Selection Avoided (b) Selection Required

Figure 3.11: Constraint assertion and FSC selection. When a new constraint asserted does not shrink the convex-hull (a) than the FSC selection step can be avoided. Whenever instead the convex-hull is shrunk (b) than the dynamic goodness of some FSC could be changes and thus a new selection is required.

III STEP – Run-time FSC selection

This is the last step of the proposed optimization strategy and its goal is to identify the optimal FSC according to the set of explicit constraints which are active time to time. Thus this is the most frequently executed step because it must run almost every time a new constraint is asserted. It is important to stress the "almost" adverb, since it could effectively happens that in consequence of a new constraints it is not actually required to select a new FSC. This step is required once the convex-hull shrank.

The assertion of a constraint not always correspond to a shrinking of the convex-hull. Indeed, in our specific context the definition of solution space, given on Par. 3.5.2 on page 85, defines the convex-hull to be the smallest convex polygon that contains just all the RFSC, regardless of the active constraints. We could identify at least two different scenario on which the assertion of a constraint does not shrink the convex-hull:

a) the assertion of an additive constraints when the convex-hull identifies a region which is already smaller that the one defined by the explicit constrains

b) the assertion of a restrictive constraint which is less binding than the one previously asserted

For example, considering the scenario depicted in Fig. 3.11a, the first case happens happens when the constraint v_1 on the additive SWM p_1 is being changed to be v_2 but without actually modifying the geometry of the convex-hull. The second

case instead is when a new constraint v_4 is asserted on the restrictive SWM p_2 where another constraint v_3 is already asserted, but with the new one which is less restrictive than the one already present.
In both of these cases, the net effect is that the set of valid FSCs is not modified and thus the convex-hull still remain the same. In this case, there is no needs to select a new solution because the one presently selected still remain the same.

In all others situations, the assertion of a constraint produces a modification of the convex-hull and thus, to preserve the empirical equivalence with the LP theoretical formulation, it is required to run the FSC selection step. Actually, even in this case we could identify two different on which the assertion of a constraint does shrink the convex-hull:

c) the additional constraint completely invalidate the previously selected FSC

d) the additional constraint only reduce the previously selected FSC

For example, considering the scenario depicted in Fig. 3.11b, the first case happens when the constraint v_1 on the additive SWM p_1 is being changed to be v_2 which completely invalidate FSC$_1$. The second case instead could be when a new constraint v_4, asserted on the same metrics as the previous one, produces only a restriction of the FSC$_1$. This time the convex-hull is modified but the current selected configuration, i.e. FSC$_1$, is not invalidated. What happens in this case is just that the dynamic goodness of FSC$_1$ change and this value, instead of its static goodness level, must be considered in order to find the next optimal configuration.

Finally, we have to observe that similar reasoning to that done so far could be done also in the case of constraints removal. In these situations: an enlargement of the convex-hull could be originated when previously invalid FSCs turns out to be valid again.

In the light of these consideration, we understand that FSC selection is a quite tricky subject, with many different conditions to be considered in order to satisfy the empirical equivalence. However, I demonstrated that the problem could be tackled and moreover quite efficient deterministic solutions can be implemented exploiting the FSC ordering provided by the previous step. Indeed, the partial ordering, once a FSC is invalidated, allows to limit the subspace to be explored near the old solution in order to find the next optimal configuration.

Chapter 4

Results, Conclusions and Developments

"The scientific man does not aim at an immediate result. He does not expect that his advanced ideas will be readily taken up. His work is like that of the planter – for the future. His duty is to lay the foundation for those who are to come, and point the way. He lives and labors and hopes."

Nikola Tesla

THIS chapter reports experimental results which show the viability and effectiveness of the power and performances optimization technique presented in this thesis. Then, it draws the conclusion of the work, and it discusses some directions where the research could extend.

4.1 Results

The major advantages of my optimization technique are the hierarchical control model provided and the low-overhead it introduces, at the expense of slightly modifications required for drivers and possibly for platform-code and user-space. The technique proposed satisfy the main requirements of modern power and performances optimization framework.

As far as the 'system-wide', 'fine-details' and 'dynamic' requirements are concerned, a quantitative comparison with other approaches is difficult to setup and bears little meaning. However, the proposed technique, thanks to its cross-layer design with multiple optimization policies at different abstraction levels, ensures to satisfy these functional requirements. As fare as 'scalability' and 'low-overhead' requirements are concerned, I have performed a complexity analysis of the optimization algorithms and a set of measurements on real hardware to evaluate these metrics.

Although the technique is general and completely independent from any architectural detail, I chose one specific mobile multimedia platform and, therefore, a specific architecture for model validation. As a target system, I have employed an NHK-8815 [99], a development board using the STn8815 SoC by STMicroelectronics. This platform has been choosed becasue it is quite representative of new generation embedded mobile platforms, being already employed on commercial products such as the recently released N96's Nokia smartphone. Actually, this platform provides support for many hardware optimization techniques and the porting of a complete GNU/Linux system is also available.

4.1.1 Worst-case analysis

A benchmarking activity as followed the implementation of the framework. This activity has been supported by a synthetic use-case, which has been properly designed to support a *worst-case analysis*. This worst-case analysis has been obtained using both a carefully designed system configuration, in terms of devices working regions, and a set of properly defined run-time requirements. These two elements allow to always stress the main algorithms implemented; e.g., the FSC search algorithm must perform a complete exploration of the configuration space, or the FSC selection algorithm always has to analyze all the available configurations to fine the one to enable.

The measurement of the time overheads, introduced by the execution of main CPM algorithms, aims at validating the complexity analysis of the model described in Sec. C.2 on page 155. Moreover, it is important to have a real overhead measure for these metrics because we aim at embedding the framework into a real product. Thus, just an asymptotic analysis is not sufficient but instead it is required real values to understand:

- how the optimization framework could impact on system performances at different time-frames (i.e., boot-time, wakeup-time, idle-time, processing-time)
- what is the corresponding real energy overhead that define the risk/benefits trade-off for the system-wide optimization policy.

These results could be used also to evaluate the scalability of the proposed solution.

Moreover, the total memory footprint of the framework's core has been measured, which is an important metrics especially when the software must be deployed on embedded systems that have reduces storage resources.

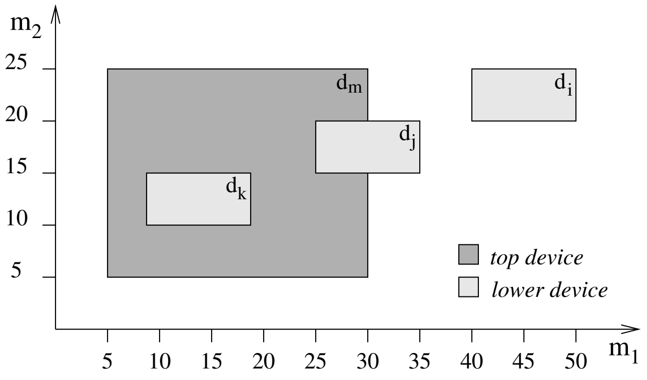


Figure 4.1: The working region defined by the test module. This module allows to configure different testing scenarios by properly configuring some parameters for some virtual devices: a top device and one or more lower devices.

The synthetic benchmark

The synthetic benchmark is composed by a kernel test module which can emulates a system with a complete set of devices, each one with its own DWRs. The number of devices and their DWRs can be properly configured at module load time to simulate different configurations and working scenarios under which the framework implementation has to be tested.

A graphical representation of the device working regions created by the test module is depicted in Fig. 4.1. The module create exactly on "top device" and one or more "lower devices". The former defines just one working region (i.e., d_m) while lower devices can define three different kind of working regions that correspond to all the possible different overlapping: full-merge (i.e., d_k), partial-merge (i.e., d_j) or not-merge (i.e., d_i). More precisely, each synthetic benchmark can configure different scenarios by loading the test module with the following parameters:

- N, the total number of platform devices to register, which corresponds to the depth of the tree on which the depth first search is performed. $N - 1$ devices are defined as "top devices" which have all the same DWR and determine the number of levels of the tree except the last one.
  Having all the same DWRs the intersection is guaranteed and avoids pruning of search path until the last level of the tree.
  The remaining device is defined as "lower device" and stands at the last level of the search tree.

- M is the number of identical DWRs of the "top devices". It corresponds to the width of the search tree.

- K is the number if identical DWRs declared by the "lower device" which fully merge with the DWRs $\in M$. The intersection of these DWRs with those of the "top devices" is complete.

- J is the number if identical DWRs declared by the "lower device" which partially merge the DWRs $\in M$. The intersection of these DWRs with those of the "top devices" is partial, they are placed across the edge of a DWR $\in M$.
- I is the number if identical DWRs declared by the "lower device" which do not merge with the DWRs $\in M$. The intersection of these DWRs with those of the "top devices" is the empty set, they are placed outside the borders of a DWR $\in M$.

The sum of K, J and I is the total number of "lower device"'s DWRs. The complete set of module parameters is then given by:

$$< N, M, K, J, I >$$

A simple bash script has been used to automatize the testing of different scenarios by loading the module with proper parameters, running the required benchmarks and collect back measurements. Each run measure the time required for the execution of each main algorithm, with a *nanoseconds* resolution, using the API offered by the standard high-resolution timer Linux framework. An average value on these measures is computed out of a pool of 30 measures for each run of the benchmark. Details on the measures and the obtained results are gathered in specific tables and then plotted to show the minimum and maximum value, together with the average value.

FSC Identification

The FSC Identification algorithm is sensible to both the number of devices and the number of DWRs that each device defines. The function defining the complexity of the algorithm and returning the number of FSCs to be identified is:

$$FSC_{count} = M^{N-1} \cdot (K + J) \tag{4.1}$$

The test module is configured with $M = 3$, N ranging from 2 to 9 and $K + J = 3$.

The results achieved are reported in Tab. 4.1 and plotted in Fig. 4.2 where both the scales are logarithmic and we show the bisector with a dashed line to determine the linear growth, useful for comparison with the trend of the measures. This comparison highlight the exponential trend of the function, which validates the complexity analysis performed on the FSC Identification algorithm. It is also relevant to note how the algorithm is fast in identifying all the FSCs of the system, especially considering a reasonable number of total FSCs of a real system that according to our analysis never reach a number greater than few thousands. With 500 FSCs the time spent by the algorithm, running on the reference platform, is less than 10 milliseconds which is at least two orders of magnitude less than the more aggressive boot-up time requirement. Thus, since that algorithm is run seldom and at boot time, the total overhead is negligible.

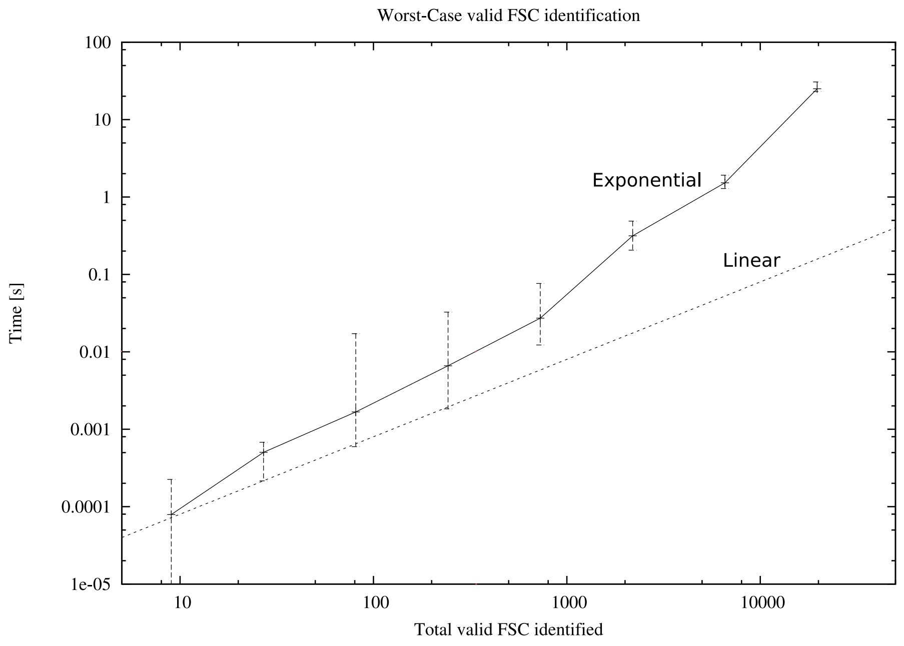


Figure 4.2: Worst-case time required by the FSC identification algorithm. Despite this algorithm has an exponential time complexity, the identification of 1000 feasible configurations takes less than one second. This is an interesting absolute value, specially if we consider that this algorithm runs just one time.

FSC Selection

The FSC Selection algorithm is sensible on the total number of FSC, which in the worst case must be all scanned. Thus, considering the Eq. 4.1 on page 102 we configured the test module with: $M = 1$, $N = 2$ and the sum $K + J$ tacking value from a predefined set of interesting configurations.

The benchmarked configurations, along with the time measurements obtained on a set of thirty run for each one of them, are reported in Tab. 4.2. These results are graphically represented by the plot in Fig. 4.3; as expected the algorithm has a linear trend. It is important to notice that, looking at absolute values, the selection algorithm is three order of magnitude better than the identification ones, with a scanning time of few milliseconds for a system with more than ten thousand feasible configurations. This quantitative result is especially important for two reasons:

- at run-time, the selection algorithm runs more frequently than the identification ones[1]. Thus, having a lower absolute overhead associated to it confirms one of our fundamental goals: run-time efficiency. Indeed, the decomposi-

[1]We already observed that, especially on embedded system, the identification is required just one time at system boot.

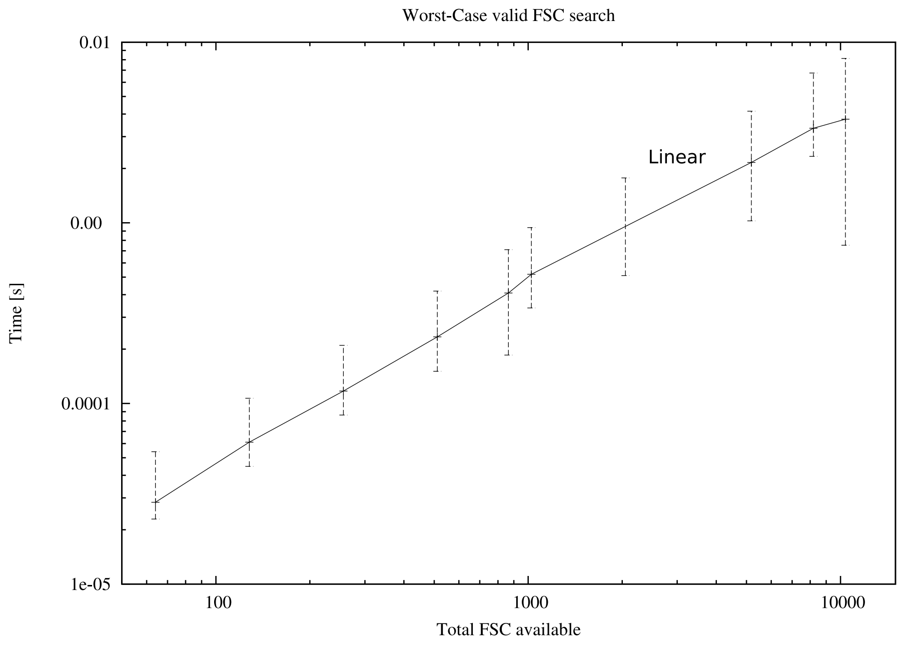


Figure 4.3: Worst-case time required by the FSC selection algorithm. The selection time never exceed the few milliseconds boundary. This is an interesting absolute which is around three order of magnitude less that the usual activation time of this algorithm.

tion of identification and selection steps, and their relative overheads, allows to spend less time (i.e., less energy) for the more frequent operations that are related to the investigation for optimization opportunities, while keeping complex (i.e., energy demanding) operations relegated to the system boot time only.

- the absolute value of the selection step is always contained within few milliseconds also for a very big number of FSC[2]. If we consider that a reasonable timeframe for the activation of a new selection is usually related to that of a use-case change at application level, we understand that this is a good result by itself. Indeed, even on a heavy loaded multi-functional embedded mobile system, it is reasonable to expect that the use-case changes could happen only every few seconds. Thus, the relative overhead of this algorithm is at least two or three order of magnitude less than that of the system under control. This has a beneficial effect on the risk-vs-benefits trade-off, increasing a lot the change to effectively exploit the proposed solution for the system-wide optimization.

[2]Even considering the relatively low performance profile of the target board used for the measurements.

# of FSCs	Samples	Max	Min	Sum	Mean	Standard Deviation	Variance
19683	30	30.6317	22.9636	751.387	25.0462	1.6082	2.5863
6561	30	1.91183	1.28636	45.7958	1.52653	0.107813	0.0116236
2187	30	0.487002	0.205745	9.45301	0.3151	0.0645083	0.00416132
729	30	0.0763771	0.0122521	0.8177	0.0272567	0.0166855	0.000278406
243	30	0.0326188	0.00183322	0.198173	0.00660576	0.00635898	4.04366e-05
81	30	0.0172099	0.000596057	0.0501677	0.00167226	0.00294663	8.68265e-06
27	30	0.000679248	0.000215581	0.0151224	0.000504079	8.79246e-05	7.73074e-09
9	30	0.000224401	1.0071e-05	0.00237652	7.92173e-05	4.34445e-05	1.88742e-09

Table 4.1: Statistics on collected data for the FSC identification benchmark. All time measures are in seconds.

# of FSCs	Samples	Max	Min	Sum	Mean	Standard Deviation	Variance
10368	30	0.00812871	0.00075171	0.112349	0.00374496	0.00153446	2.35456e-06
8192	30	0.00675012	0.00233277	0.100079	0.00333597	0.00106858	1.14186e-06
5184	30	0.00415022	0.00102456	0.0647618	0.00215873	0.000768907	5.91219e-07
2048	30	0.00177145	0.000510312	0.0286649	0.000955498	0.000365238	1.33399e-07
1024	30	0.000941163	0.000337806	0.0155594	0.000518647	0.000219977	4.83898e-08
864	30	0.000709989	0.000185277	0.0122633	0.000408778	0.000163383	2.66942e-08
512	30	0.000418883	0.000150714	0.00700969	0.000233656	0.00010086	1.01727e-08
256	30	0.000209465	8.6241e-05	0.00351589	0.000117196	4.73157e-05	2.23877e-09
128	30	0.000106776	4.484e-05	0.001833	6.11e-05	2.30962e-05	5.33432e-10
64	30	5.4068e-05	2.2894e-05	0.000850984	2.83661e-05	8.18131e-06	6.69338e-11

Table 4.2: Statistics on collected data for the FSC selection benchmark. All time measures are in seconds.

4.1.2 A safari on a real-world usage scenario

The aim of the following discussion is to show the reader how CPM can be easily and effectively adopted in a real-world system. A careful reader should catch the benefits of using CPM to manage resources, such as the Internet connection bandwidth, according to the actual applications demand. Moreover, a system integrator should evaluate how simple and straightway is the effort required to keep track of architectural dependency between different subsystems, while still preserving the opportunity to write a clean code that exploit all already existing low-level optimization frameworks.

The STn8815 SoC [99] has an ARM host CPU and two DPS accelerators for multimedia: one for audio transcoding (i.e., DSP_A) and another supporting video accelleration (i.e., DSP_V). The host CPU is clocked by the CPU_CLK clock signal, while both the two DSPs are clocked by the same DSP_CLK clock signal. Interestingly, this SoC is characterized by a strict dependency between CPU_CLK and DSP_CLK, which constrains their frequency. When one or both the DSPs are active and require a certain clock frequency, the CPU is constrained to work at a compatible frequency. This means that the CPU frequency scaling driver, which in a standard Linux kernel is provided by the CPUfreq framework (see Par. B.2.2 on page 143), must be opportunely notified about the hardware inter-dependency and its policy constrained. It is worth to notice that, simply hacking the CPUfreq framework, to take care of the hardware interdependency, don't properly solve the problem because such a solution is not portable and it cannot even be accepted into the mainline kernel.

From the user-space standpoint, the use case involves an application that controls the playback of a audio-video stream, and another one that downloads some data from the web, such as a download manager handling some queues or a mail client fetching new messages. These applications share a resource, the connection bandwidth, and require a minimum amount of it to grant the wanted QoS. The devices involved in the use case are a 3G modem which provide Internet access, the two DSPs used for hardware accelerated audio and video decoding, and of course the CPU running both the OS and the two aforementioned applications among others.

The modem provides several working modes, one for each of the mobile network's protocol supported. Each working mode is characterized by a maximum bandwidth capacity and a different level of energy consumption. The audio and video DSPs provide different codecs such as, respectively: raw PCM, MP3, WMA and OGG-Vorbis for the audio accelerator, and MPEG4, H.263 and H.264 for the video ones. Each hardware accelerated decoder requires different operational frequencies. Finally, the CPU is controlled by the CPUfreq framework [12] which controls the processor's operating voltage and frequency according to the computation load. Notice that the CPU load is defined only by the OS and running applications, but not by the audio and video decoding which instead are activities in charge of the two DSPs.

The SWM reasonable and thus considered for this usage scenario are:

- *the connection bandwidth*: represented by the additive metric 'BAND' that abstracts the network bandwidth's resource on which applications compete for usage;
- *the audio codec*: represented by the restrictive metric 'ACODEC' that abstracts the kind of audio content that an application has to decode;
- *the video codec*: represented by the restrictive metric 'VCODEC' that abstracts the kind of video content that an application has to decode;
- *the clock signal of the accelerators*: represented by the restrictive metric 'DPS_CLK' that abstracts the frequency of each DPS's clock signal;
- *the CPU clock signal*: represented by the restrictive metric 'CPU_CLK' that abstracts the host processor's frequency.

The first three are abstract metrics (i.e., ASM), thus exposed to applications for the assertion of QoS requirements, while the last two are platform metrics (i.e., PSM) that are defined by the platform code and used internally by the drivers only.
Using the available system wide metrics, each device driver defines its own working regions as graphically depicted in Fig. 4.4-abc. The architectural dependencies instead are defined by the Linux platform initialization code and have been represented in Fig. 4.4d.

The use-case begins with the user selecting a content to be played. As soon as the download of audio and video data starts, the video player application collect some information:

- v_1: the minimum connection bandwidth (e.g., 264 Kbps) required to have good quality reproduction, i.e. without jitter and buffering phenomena;
- the audio codec used (e.g., MP3);
- the video codec used (e.g., H.263);

and use them to assert some QoS requirements on the corresponding ASM, respectively: BAND, ACODEC and VCODEC.

The CPM core collects and aggregate these requirements to find the corresponding constraints on every system metric. At this point, the current FSC the system is working on could be invalidated by the new constraints. Thus, the FSC selection algorithm is run to fine the next valid system-wide feasible configuration.
Once the new valid FSC has been found, the CPM framework notify the corresponding DWRs to each involved driver along with the values of the new system constraints. Since the required audio and video codecs are bound to a specific DSP_CLK frequency and the platform DWRs defines its dependency with the

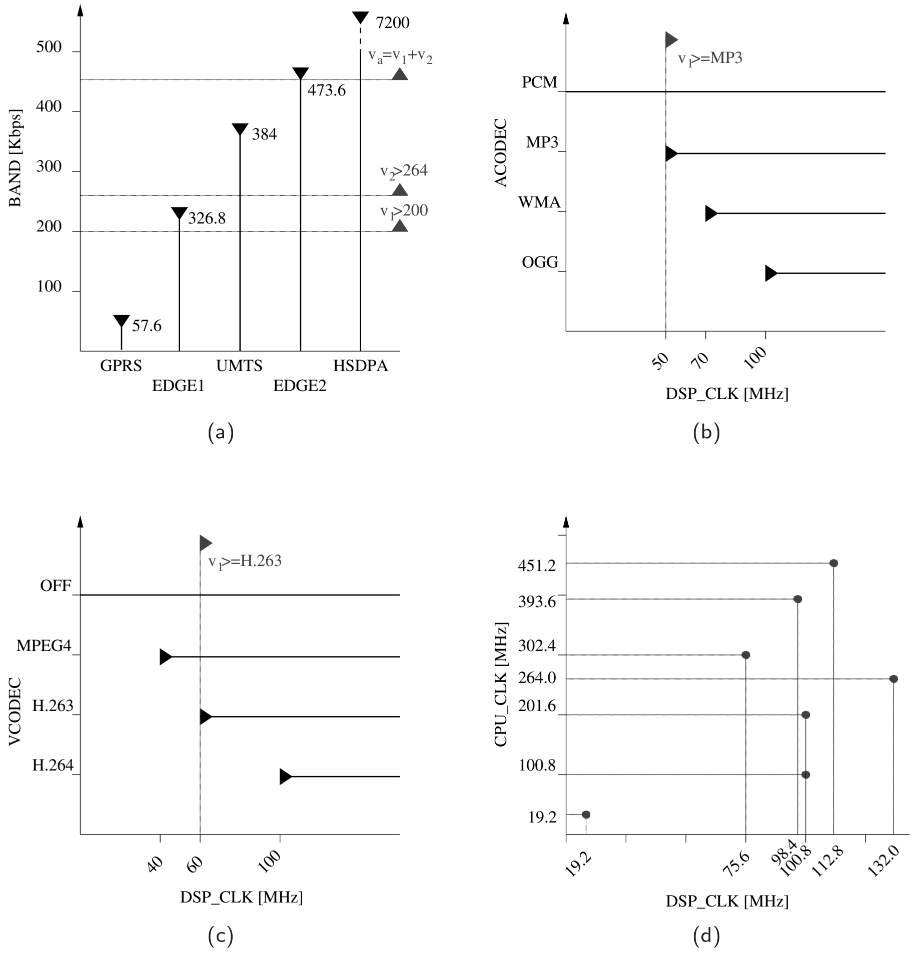


Figure 4.4: The set of device working regions defined by the usage scenario. a) modem's DWRs, with the lower bound requirements v_1 and v_2 asserted by applications and the corresponding aggregated constraint v_a. b/c) audio/video DSP's DWRs which setup a relation between the required audio/video decoding effort represented by the ACODEC/VCODEC metrics and the hardware accelerator clock frequency abstracted by the DSP_CLK metrics. d) platform DWRs to track the architectural dependency between two metrics: CPU_CLK and DSP_CLK

CPU_CLK, then the CPUfreq framework will be able to scale the frequency according to the imposed constraint. Also all other involved subsystems update their working mode accordingly, for instance the modem switches from the GPRS to the EDGE1 network to fit the new constraint for the BAND metric.

While the video is played, the data download application starts and asserts a QoS requirement on the bandwidth for a minimum amount of 200 kbit/s. Since the BAND metrics is of additive type (ref. "composition" on pag. 67), the CPM framework perform an aggregation by summing the new requirement with the old constraint's value to get the new ones (i.e., 464 kbps). This new constraint invalidate one more time the current FSC and thus the FSC selection algorithm is triggered one more time to move the system towards the next valid system-wide feasible configuration. This new configuration specifically brings the modem device to move to a different working mode (i.e., EDGE2).

Use case results and considerations

From this use-case it is possible to derive some main consideration on the proposed framework.

System resources management. The usage of QoS requirements' declaration and aggregation allows to keep a correct and precise view of used and still available system resources. This information is exploited to configure the hardware devices with the correct working mode supporting the required resource demand. It is worth to notice that the usage of an efficient FSC selection algorithm allows to achieve the better energy saving that is compatible with the user perceived performances. Indeed, this algorithm is always working on a set of possible system-wide solutions which have been pre-ordered according to the running performance-vs-power optimization policy. This solution is better than the best effort approach provided by many of the current implementations of system-wide resource management systems, such as the QoSPM framework described in Par. B.3.2 on page 151.

Additive aggregation. The additive aggregation is a concept introduced by CPM to overcame some limitation of present implementations such as the QoSPM framework. According to this framework, even for resources that are intrinsically additive (e.g., bandwidth) the aggregation function could be only of type 'min/max'. This do not allow to keep a correct view of system resources and could bring to the selection of devices' working mode that can't support the actually required QoS level. For example, if two applications require 300 kbps each, QoSPM aggregating with the 'max' function will still set a system-wide constraint at only 300kbps, which is only half of the effectively required bandwidth resource of 600kpks.

Dependency tracking. CPM allows to track hardware dependencies, among different subsystems of a SoC, in order to prevent a correct operation of a system. Instead of patching each device driver to adapt it to a different platform, system developers must simply declare platform DWRs to track the dependencies' issues. That way code portability is improved allowing to have a single driver which can safely fits multiple platforms with different architectural requirements.

Automatic identification of system working points. Other approaches to power management, like the DPM framework described in Par. B.3.1 on page 150, requires to code all the working points by hand. Instead, CPM allows to identify all the working points of an entire platform automatically through the computations of the FSC in the system configuration phase, e.g., at boot-time. It does this exploiting the information defined, independently, in each device driver code. This is another relevant result, since it improves portability of device drivers code across different platforms and products.

4.2 Conclusions

This thesis describe a methodology to identify the optimal trade-off between perceived performances and energy consumption of a multimedia mobile embedded system, considering both application requirements and system resources.

The proposed method is formally verified and supported by a complete implementation within a Linux kernel framework named CPM.
Thanks to an optimized implementation, both user-space applications and drivers could get immediate benefits from its usage without noticeable performances degradation. Indeed, it proves to be effective on identify the optimal configuration, according to a give set of optimization objectives, and to exhibit a very low overhead on real utilization scenarios.
Moreover, supporting the effective implementation of a distributed control and hierarchical optimization solution, it is able to transparently handle devices interdependencies and architectural constraints. Indeed, the proposed solution offers an easy to use interface to both system and application programmers. For instance, the framework could also support a regular resource manager with a very limited integration effort.

Finally, the design of the proposed technique and the modular architecture of its implementation effectively support code reuse and thus improve the portability of the solution on different platforms.

4.3 Developments

This thesis work paves the way for a wide set of improvements and future developments.

Extending hardware support

Up to now, just a couple of use-cases has been developed and implemented on real hardware. The main goal was that of testing the implementation and proving the goodness of the proposed approach in different scenarios, especially if compared with other available Linux frameworks like QoSPM [100].

The framework implementation can be used as an OS based simulation environment, running on real hardware or within a virtual machine. This usage is useful to easily evaluate the distributed control model and how well it could interact with different local optimization policies. However, this framework is designed for real use and thus a wide adoption within different hardware platforms is one of the main targets to definitely prove its goodness.

Porting the core, policy and governors, as well has testing drivers is a trivial task since CPM has been directly coded on a recent Linux kernel: if the board already comes with a recent kernel, the porting of these components is just a matter of simple cross-compilation. If the board instead comes with an older Linux version, it is possible that some kernel-related data structure used by CPM does not match with our implementation, thus a minor porting effort should be considered.

The main effort in integrating CPM in a real hardware platform is represented by already existing drivers, that should be modified to support the framework's workflow, and the corresponding identification of DWR.

SWM mapping methodology

A further improvement of the current work consists in the definition of a clear and standard methodology to perform the mapping of SWM on device drivers operating mode. For instance, given a specific device and the analysis of its internal configuration states, how a developer should generate the mapping and the local driver's policy. This is a key point for the success of CPM and its widespread adoption.

Improvements at applications level

A complete usage of the proposed framework includes the participation of user-space as an active entity. Applications should assert their QoS requirements that are used to constraint the identification of feasible configurations. Up to now, only a trivial interface towards user-space has been developed, to further improve the utilization of the framework these extensions are considered particularly interesting:

- *Improved API* – The user-space should be able to interact with the in-kernel framework implementation using an efficient communication mechanism, perhaps based on a properly defined syscall interface.

- *Feedback mechanism* – User-space should be able to receive feedbacks on the current system resource availability and granted QoS. This could allows for applications to fine tune their offered services according to effectively available resources and perhaps also to tune these requirements at run-time according to changing resources availability.
- *Priority aggregation* – Requirements coming from applications should be managed according to their impact on the user's perceived QoS. Thus, applications with higher impact on user experience should be granted higher priorities on constraints assertion, regardless of the order these constrains are asserted within the system. A mechanism to support this kind of feature must relay on a properly working feedback mechanism previously outlined.

The improvements described above depict CPM as a resource manager and outline a possible limitation of the ideas presented: applications should be modified to support CPM. This is not a good feasible solution because it would introduce an unsustainable effort in the user-space context. This limitation could be easily overcame thinking at integrating CPM with software layer that stands between CPM and the applications. This mid-layer is in charge of managing the bi-directional communication between the other two layers. Resource managers already exists: thus the integration of CPM with one of such framework is foreseen as an interesting extension of this work.

Runtime power measures

To better support power optimization it is interesting the implementation of the learning mechanism to acquire power consumption measures and used them to automatically power classify each FSC. To implement such a mechanism it is required first the porting of CPM on a board with sensors or dedicated PMU to collect energy consumption measurements.

Governor modules

The development of new Governor's modules for the FSC identification is another interesting development topic. Actually, as the experimental results prove, the algorithm based on an exhaustive search, even if it has exponential complexity in the worst case, performs very well and introduce a really tiny computation overhead on the system. Especially if we consider that such algorithm is run only at boot time. It would be interesting to better investigate the possibility to implement some kind of hardware acceleration to support the main complex framework functionalities.

Appendix A

An Implementation Proposal

"Talk is cheap. Show me the code!"
Linus Torvalds

THIS appendix describes the CPM framework, which is a reference implementation for the proposed optimization technique. The main design decisions and some implementation details will be carefully reviewed in order to setup a common ground for everyone interested on using or hacking the code.

A.1 Theoretical Concepts Implementation

CPM has been developed following the theory of Hierarchical Distributed Control. However, some of the theoretical concepts have been implemented slightly differently to ensure both portability and run-time efficiency. In the following paragraphs some theoretical ideas are recalled and, for each of them, the various choices made are explained and justified.

A.1.1 The System-Wide Metrics (SWMs)

The implementation of SWMs reflects the definition given for these elements in Par. 3.3.1 on page 65, while describing the theoretical model. They are abstract metrics that represent different aspects of the global QoS level of a system. Device drivers can be sensible to different subsets of SWMs. Through these metrics, drivers are kept informed of the global system state, they can use these informations for taking their local decisions and they can assert constraints on them when, for working correctly, they need to be ensured on specific levels of QoS.

The main issue that I found during the implementation of the SWM's concept is the definition of a set of them. This topic has been analyzed considering the typical characteristics of SoC based embedded systems, for which CPM is mainly targeted. Every embedded systems has its own characteristics and functionalities and so it appears impossible to define a unique static set of SWMs suitable for all of them. For this reason I decided to statically define a list of platform-independent metrics, ASMs, directly within the CPM core, while I provide a mechanism that allows the system platform code to define additional platform-specific metrics, PSMs, which can be used to represent find-detail about a specific platform. Since ASMs are platform independent, they are exposed to user-space. To the contrary, the PSMs will be visible to the platform code and drivers only. This decision is meant to preserve the portability of the framework.

A.1.2 The Device Working Regions (DWRs)

The implementation of this concept completely covers the definition introduced in the Par. 3.3.3 on page 70. A DWR express the binding between an operating mode of a device and the corresponding QoS levels of the system. They are represented by ranges on a subset of SWMs and defined by the corresponding device driver.

A driver propose a change on a SWM when its internal configuration changes, for instance because the local policy of the driver can optimize the operating point. In this case the driver affect the SWM. Similarly, a driver can be asked to change its configuration to satisfy a QoS requirement, asserted by another entity (either an application or a different driver). In this case the requirements must constraints the value on one of the device's sensible SWM. Whenever a driver changes its internal configuration the corresponding DWR could also be updated.

DWRs are essentially static entities: they are related to the possible devices' configurations and, consequently, to their physical capabilities. It is therefore reasonable to assume that they can be statically known by device drivers and that they can not change at run time. The device will be also considered both influenced and affect by all the SWMs that are used to define its DWRs. Due to these reasons, to ensure that CPM works correctly, devices must declare all their DWRs at registration time using a specific API defined by the framework. All the DWRs are stored by the framework for future elaborations: like the computation of FSCs or the notifications during the distributed agreement process.

A.1.3 The Feasible System-wide Configurations (FSCs)

According to the definition given on page 78, the FSCs as the regions in the SWCS that result from the intersection of at least one DWR for each device. These are the only feasible working regions of the entire system. The problem of a possible high computational cost of FSC search is investigated in Sec. C.2 on page 155 and proved in Sec. 4.1.1 on page 100. These configurations are strictly related to the defined DWRs that, as explained, are immutable for each device once the driver is compiled. This implies that FSCs can change only when a driver registers or unregisters its devices to the framework and, consequently the process of FSCs identification has to run only in these two situations that are relatively seldom in embedded systems. As a result, the probability of a relevant overhead is reduced if all drivers register at boot time: overhead appears only once, at system initialization.

A.1.4 The objective function

An important part of the hierarchical distributed control theory is related to the analysis of the problem through an approach that is based on linear programming. This kind of strategy requires the definition of an objective function to be used for choosing the system working point, which corresponds to one of the available FSCs. The selected FSC, besides satisfying all the asserted constraints, results to be the best according to the considered optimization goals. In linear programming an objective function is represented by an oriented vector that indicates the direction to take to find the best solution inside the constrained solution space.

Within the implementation, I reworked the concept of oriented-vector which defines the objective function into something that is easier to implement and manage with the kernel data structures. I considered that a vector in a n-dimensional space can be always represented as the sum of its n projections on the n direction that generate the space. According to this consideration, inside CPM I provided the mechanism to define these n projections. This mechanism simply corresponds to the assignment of a 'weight' to each SWM. These weights allow to evaluate each FSC according to the same approach suggested by linear programming.

A.2 The Framework Architecture

An overview of the proposed Linux kernel framework implementation is shown in Fig. A.1, where all the main components are represented to highlight also their relationship. The framework's implementation is made of different interacting components which, according to their logical role, can be divided into three main classes

- CPM framework
- Device drivers
- Platform code

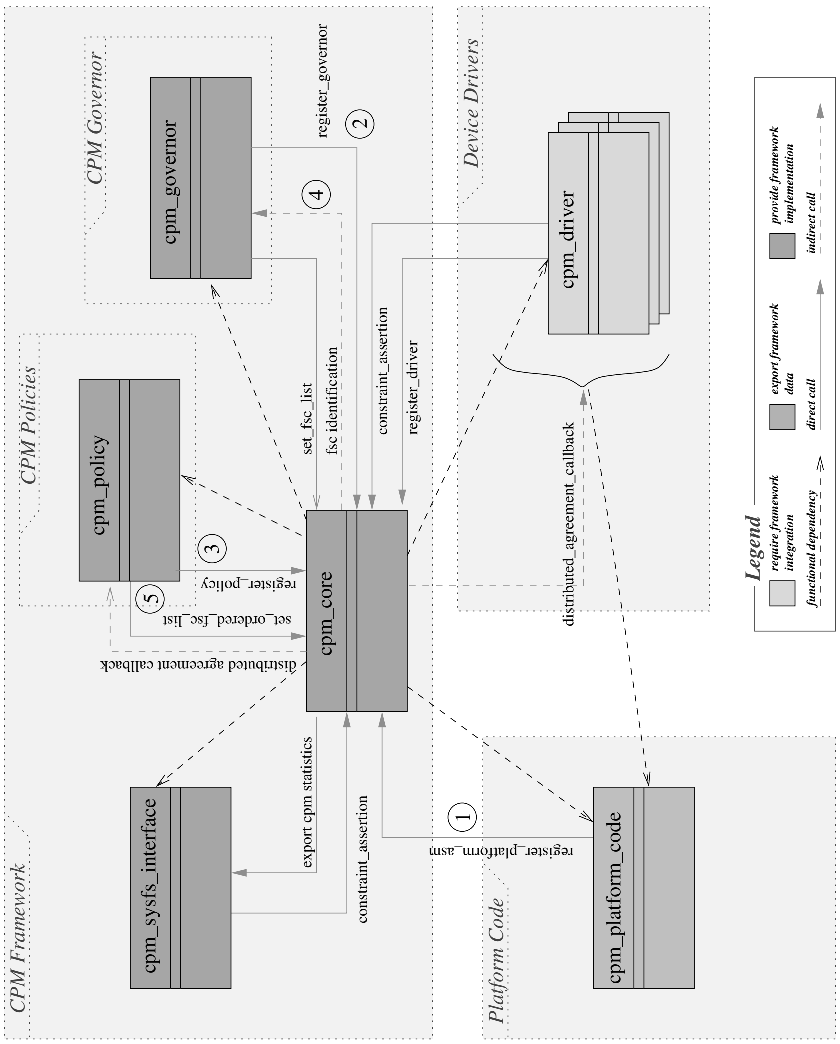


Figure A.1: Overview of the CPM framework architecture. This block diagram represent the framework's components and the relationship among them. There are three main class of components: framework core, platform code and driver. The last two define code that is already available in the operating system and must be just integrated. The platform code defines the PSMs, while the drivers define their DWRs. Operations from one to three usually happens at boot-time, while operations four and five happens only when the framework is enabled. At run-time, either drivers or applications (using the sysfs interface) can assert and remove constraints that could trigger a system reconfiguration.

Almost all the entities introduced by the theoretical model are defined by the components belonging the first group. The drivers define the DWR and the platform code is in charge to define the PSM. Let us review in details each one of these classes.

A.2.1 CPM Framework

This is the core of the proposed framework and is actually composed of three different modules.

The CPM core (`cpm_core`). This is the main module of the framework and contains the entire core logic. It can be primarily paired with the theoretical idea of *Constrained Power Manager (CPM)* which, according to its definition, it is the only component that is known to all the others. This module keeps track of DWRs, computes FSCs and control the workflow of information exchanged by other modules. The actual identification of FSCs and their ordering according to the objective function is demanded to other modules, respectively the `cpm_governor` and `cpm_policy`.

The core module also defines the API used by the platform code to setup the PSMs and drivers to register themselves to the core and to define their DWRs.
Finally, this module provides also a user-space interface based on sysfs. This is used to export information about SWMs, and the constraints that are currently asserted on them. Moreover, using this interface new constraints can be asserted. Indeed, this interface provides a hook to the “execution context”, which is composed by applications, libraries and software buses that describe what it is happening in the system from the user standpoint.

The CPM governor (`cpm_governor`). This module is in charge of the FSCs identification. The framework allows to define more than one governor. Each one must register itself within the `cpm_core` using the provided API. However, at any time, only one can be selected and enabled for use. Given the list of registered devices and the corresponding DWRs, a governor (i.e., the one which is active) is expected to returns the set of FSCs identified. Once registered, the identification functionality can be used by the core every time it’s required.

The choice of implementing the FSCs identification step as an independent kernel module, instead of embedding it inside the core, is due to efficiency’s and flexibility’s considerations. Moreover, separating core from governors and policies it is a well known and successful programming approach used also in other Linux kernel frameworks (e.g., CPUFreq). Indeed, this allows to test different algorithms or to tuned them in a proper way. Moreover, considering embedded system where it is possible that the set of registered devices never changes for the entire life of the final product, thanks to this modular approach, FSCs can be statically computed one time and then coded into a governor that simple export them to CPM, without any further run-time research processing required.

The CPM policy (`cpm_policy`). This module is in charge of ordering FSCs according to the implemented objective function. Even in this case, the framework allows to define more than one policy, each one registering itself with the provided API, and only one active at any time.

The decision to make this functionality independent from the `cpm_core` is related to flexibility considerations. The general idea of CPM is to evaluate FSCs according some weights defined on SWMs. Anyway, the compliance with this basic idea, does not imply the existence of an unique objective function. For example, a policy can consider only a subset of SWMs, e.g. only latencies, to optimize a specific aspect of the system. Thanks to this modular implementation, the need for different strategies can be fulfilled through either the implementation of different modules or that of a single module with some tunable parameters.

A.2.2 Device drivers

Each driver must register itself and define the DWRs of each managed devices, right after its loading into the system. The control of the local configuration is demanded to a driver's own management policy, that is usually finalized to optimize some local metrics, without considering the other component of the system. However, for a compliant integration within the CPM framework, the local policy is expected to interact with the framework before any change on the device working mode. Indeed, the policy can answering to the various notifications that the core sends to drivers during a FSC change, to express its agreement or disagreement to a proposed local reconfiguration.

A.2.3 Platform code

This code is used to set up PSMs, the platform dependent metrics, which provide higher flexibility and portability to CPM. This portability could be ensured through a binding between PSMs and ASMs. Indeed, DWRs could be expressed on both these two kind of metrics. This allows user-space to assert requirements at run-time on platform-independent ASM and thus to reflect these requirements on some other platform-specific metrics. For example, an hypothetic 'network bandwidth' ASM can become both a 'LAN speed' or 'modem network' depending on the real type of connection available on the target system.

A.3 The Framework Workflow

Two are the main timeframes in which the framework is used: at system initialization time and at run-time. In this section I clarify the role of each component and the details of the involved mechanisms during each one of these timeframes.

A.3.1 The role of CPM at system initialization

An overview of the activities related to CPM, which happens at boot time, is depicted in Fig. A.2. Among these activities, the really first fore are executed at system initialization, while the last two can start once the framework is enabled. Let me review each one in details.

PSMs registration. At boot-time the platform code is required to register all the platform specific metrics. To that purpose, the platform code calls a registration method, defined by the core's API, and passes it an array containing the PSMs. For each PSM this set of attributes is required to define the metrics properties such as: name, aggregation type, composition type and boundaries for acceptable values. Another parameter allows to define whether the metrics should be kept hidden or read-only exported to the user-space.

Governor registration. Every governor must register itself to the `cpm_core` module. This operation is usually accomplished immediately after a governor's module has been loaded. The governor's initialization function calls the proper API registration function, provided by the core, to define a reference to its FSC identification algorithm.

Only one governor can be used by the CPM framework at each point in time. If more than one governor is registered a suitable user-space interface is provided to select the one to be used by the framework. Every time a governor is changed the framework require the new active governor to identify the FSCs.

Policy registration. Similar for the governors, policies also must register to the framework using the provided API. During registration each policy passes a reference to two callbacks functions implementing the FSC ordering and the DDP monitoring algorithms. Even in this case only one policy can be active in CPM at every time. If more than one policy is registered a suitable user-space interface is provided to select the one to be used by the framework. Every time the active policy is changed the framework requires the new active policy to define the FSCs ordering.

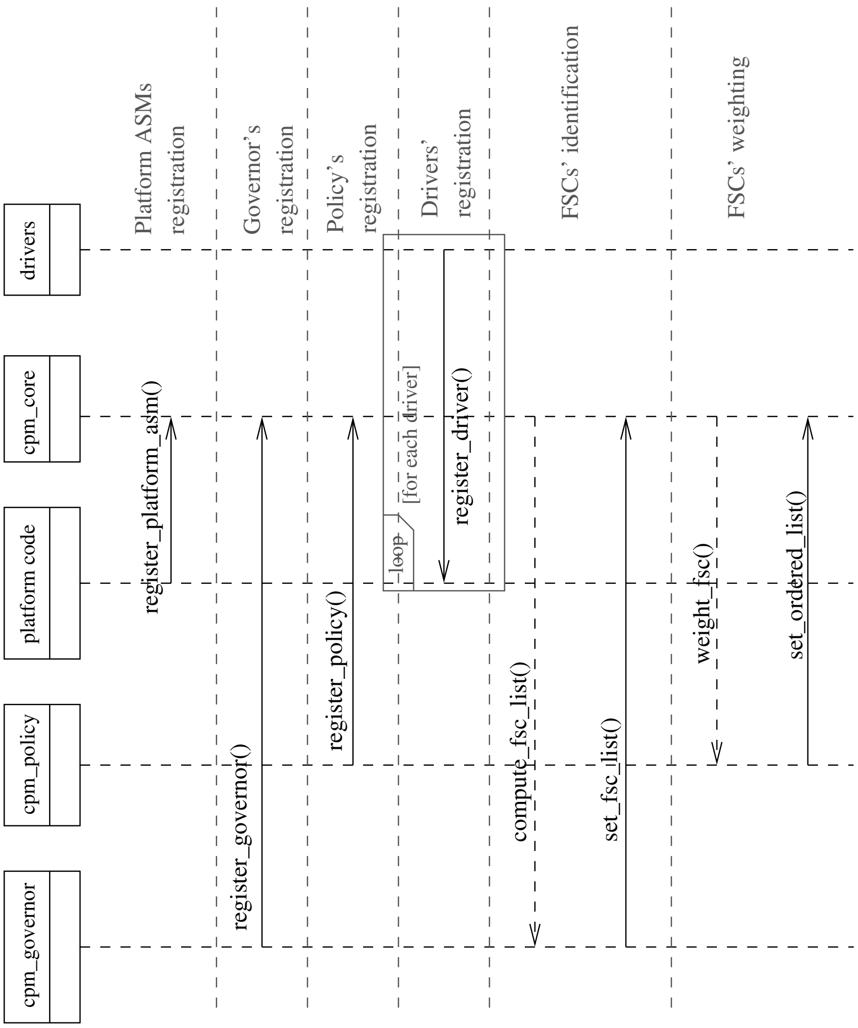


Figure A.2: A sequence diagram of the framework initialization. While the first three steps usually happens at boot-time, the last two can happens only once the framework is enabled.

Drivers registration. This is the last action which is mandatory at system initialization. The registration of drivers is possible at run-time, but especially in the case of embedded system it happens only at system boot time. Every drivers registering to the framework provides its list of DWRs and a reference to a callback function. The list of DWRs is used to infer the set of sensible SWMs of the corresponding device. The drivers will then be allowed to assert constraints only on them. The callback function is instead used to notify the device, during the distributed agreement process, about the change of the global system working point.

FSCs identification. This activity starts whenever the set of FSCs is outdated. This happens at system boot, when FSCs should be identified for the first time, or whenever a device is registered or removed from the system. Another event that triggers a new identification step is the change of the active governor.

Once all FSCs have been identified, they are inserted into a linked list and passed back to the core using a proper method defined by its API.

FSCs ordering. This activity starts every time a new policy is activated or its parameters are updated. In all these cases, it is necessary to re-order the set of FSCs that will then be used by selection algorithm. Once all FSCs have been ordered by the active policy, they are inserted into a linked list and passed back to the core using a proper method defined by its API.

A.3.2 The role of CPM at run-time

The CPM framework is mainly used at run-time, when it is in charge to support the system-wide optimization according to the application requirements and the active optimization policy. During this phase, the framework is mainly involved in activities related to requirements aggregation, constraints synthesis, search of a FSC and its notification to the driver's local policies. An overview of the main activities related to this timeframe is depicted in Fig. A.3. Let me review each one in details.

Requirements assertion. This is the start action that triggers the possibility of a system reconfiguration. Either applications, according to their requirements, or drivers, according to its local optimization policy, could need a different working mode. This need is expressed by a QoS requirement assertion on some sensible SWMs. Depending on the type of requester, the assertion can be communicated to the framework in two ways. In the case of a drivers, using the framework API, while in case of an application using the sysfs interface.

Requirement validation. The active policy is required to validate each new QoS requirement. The policy can evaluate the new requirement, considering both the requesting entity and the actual value, and than return its agreement or not to proceed. If the policy returns a failure, the research of a new FSC terminates and a failure notification is returned to the driver or to the application requiring it. The policy takes its decision according to a proper criteria. For example, the policy could enforce some access control rules on who is authorized to assert some requirements.

Requirement aggregation and constraint assertion. Once a QoS requirement has be authorized by the active policy, the framework aggregate it with those previously asserted for the same SWM. The aggregation is done according to the properties of the SWM: its composition, either additive or restrictive, and its type, either LiB or GiB, and produce a new constraints.

FSC selection. Once a new constraint invalidate the current F SC, a new once must be selected. The FSCs are stored in a list, which is maintained ordered according to the objective function implemented by the current policy. The selection of a new FSC is done scanning this ordered list and stopping at the first element that is compatible with the set of all the asserted constraints. This guarantees to choose always the best possible FSC, according to the current policy. If a candidate FSC is not found the requested constraint is not feasible and also in this case a failure notification is returned to the driver or to the application that had required it.

FSC validation. When a new valid FSC is found, it is notified to the policy that can evaluate it according to its own criteria. If the policy rejects the proposed FSC, the previous step could be tried again until either a valid configuration is authorized buy the policy or anymore FSC are available. In this last case the system reconfiguration fails and the requirement's caller notified.

Pre-change notification. Once a new valid FSC has been validated by the active policy, the system is ready to switch to the new configuration. However, before the actual switch, all devices are notified to give their local policies the final chance to stop the change's process. Indeed, by returning a not agreement a device local policy could require the seek of another FSC. Otherwise, when a devices agrees on the proposed FSC, it must be ready to reconfigure itself according to the DWRs defined by the new FSC. When all devices have returned their agreement the policy is notified about the end of the pre-change phase.

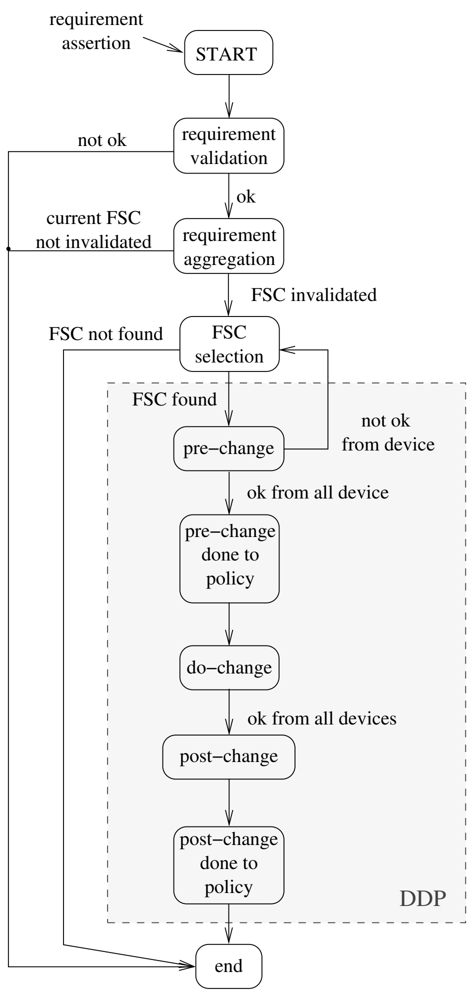


Figure A.3: The state diagram describing the behaviors of the framework at run-time. The CPM core provide support for QoS requirements assertion, validation and aggregation. The aggregation procedure could produce a new constraint on the system-wide metrics that trigger the selection of a new FSC. The selection, agreement and activation of a new FSC are part of a distributed decision process (DDP) which ensure an agreement between all the local control policies before changing the system-wide configuration.

Do-change notification. Once all devices have accepted the new FSC, they are asynchronously triggered to reconfigure themselves in the new selected FSC. This notification in an asynchronous notification that trigger the reconfiguration without waiting for it to complete. Thus, all the drivers can reconfigures in parallel the corresponding devices.

Post-change notification. This is a synchronization call. Once called, every driver should return from this notification only when its reconfiguration has completed. Thus, when all drivers return from this call the system is granted to be moved to the new configuration. One more time the policy is notified about the reconfiguration completion.

The last three activities define what we called a "Distributed Decision Process" (DDP), which is depicted in Fig. A.3. These process is particularly important because give the change to the driver's local policies to play an active role in the system-wide reconfiguration during the selection of the next valid FSC to activate. Indeed, every and only the valid FSC are proposed to every device driver to let them evaluate the possibility to move to the new configuration. Only when all the local policies agree about the new configuration, then this configuration in enabled. This allows to get a real hierarchical and distributed control.

A.4 The Framework Interfaces

The CPM framework's' API interface allows different entities to transparently interacts by accessing the framework's services according to the workflow previously described. This interface is based on a set of proper defined data structures and functions, which corresponds to the theoretical concepts of: SWM, DWR, FSC, objective functions, and constraints. Additional data structures are instead used to represent policies, governors and device drivers. All these data types and function prototypes are defined within the `include/linux/cpm.h` header file. This section deals with the description of this programming interface. A comprehensive diagram of the data structures used by the framework and their interactions is depicted in Fig. A.4.

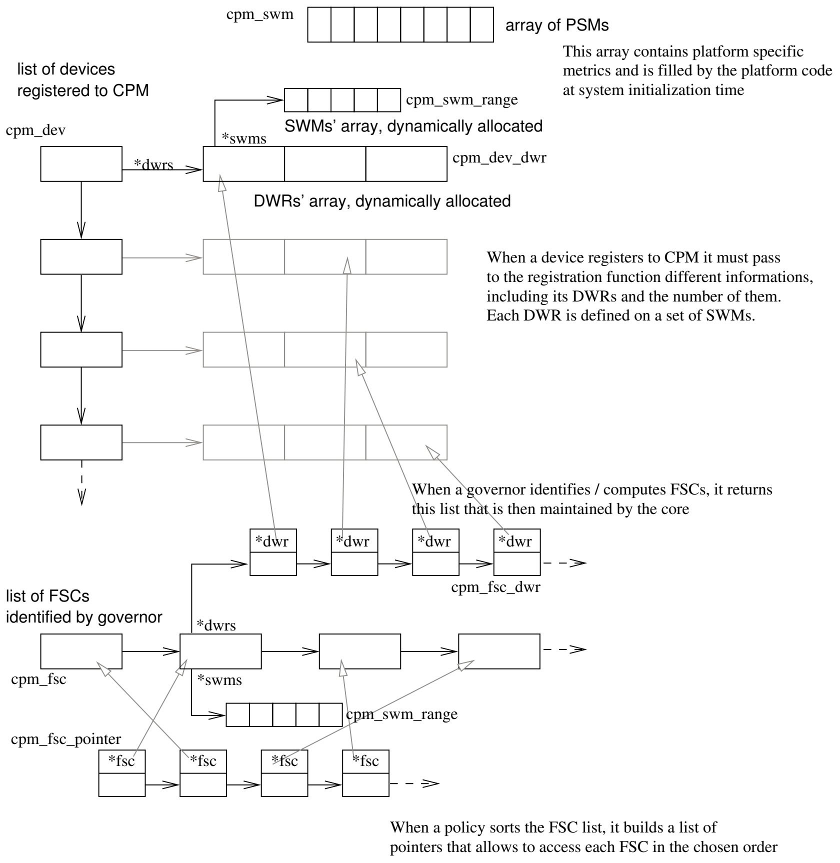

cpm_swm
array of PSMs
This array contains platform specific metrics and is filled by the platform code at system initialization time
list of devices registered to CPM
cpm_swm_range
SWMs' array, dynamically allocated
cpm_dev
*swms
*dwrs
cpm_dev_dwr
DWRs' array, dynamically allocated
When a device registers to CPM it must pass to the registration function different informations, including its DWRs and the number of them. Each DWR is defined on a set of SWMs.
When a governor identifies / computes FSCs, it returns this list that is then maintained by the core
*dwr
*dwr
*dwr
*dwr
cpm_fsc_dwr
list of FSCs identified by governor
*dwrs
cpm_fsc
*swms
cpm_swm_range
cpm_fsc_pointer
*fsc
*fsc
*fsc
*fsc
When a policy sorts the FSC list, it builds a list of pointers that allows to access each FSC in the chosen order

Figure A.4: Overview of the relationship between the main data structures. The CPM core implementation rely on statically allocated data (e.g., PSMs and DWRs) that are used to dynamically generate other data (e.g., FSCs), according to some modular code (e.g., governor and policy).

A.4.1 The basic data types

Let me first review how are defined the model's theoretical concepts, which has been presented in the first part of this appendix.

The SWM type. Each system-wide metric is represented with a data structures of type `cpm_swm`, shown in Lis. A.1. This type allows to keep track of all the information provided at initialization time: some constants definitions help to properly encode the different options. In addition to this, a field is used for the ASM's weight that is related, as previously explained, to the objective function. These structures are allocated by platform code and then passed to `cpm_core` that maintains them into an array for use at runtime.

The DWRs type. Each device working region is represented by a data structures with type `cpm_dwr`, shown in Lis. A.2. These allow to uniquely identify each DWR with an id and a reference to the device at which it belongs. For each DWR all the SWMs' ranges that compose it are stored into a dynamic array. Each range is defined by the `cpm_swm_range` data structure that is based on the `cpm_range` base type represented into Lis. A.3.

Finally this data provides some helpers for the sysfs interface management, and two additional fields that can be used by policies and governors for data that can be eventually needed for their elaborations. These structures are declared by the drivers and passed to the core when each driver is registered.

The FSCs type. The feasible system-wide configurations are stored in data structures of type `cpm_fsc`, shown in Lis. A.4. They are uniquely identified by an id and, like DWRs, are composed by a set of ASMs' ranges represented by the structure of Lis. A.3. In addition each FSC has also references to the various DWRs that have been merged, by the `cpm_governor` to obtain the FSC itself. This information is use at runtime for the notification of the new active FSC to devices. Finally, also in this case the structure provides fields that can be used by policies and governors for their own needs. These structures must be declared by the governor during FSC identification phase.

The following data structures, are used to represent framework's entities inside the `cpm_core`, mainly for keep track of their callbacks.

```
struct cpm_swm {
        char name[CPM_NAME_LEN];
        void *data;
#define CPM_TYPE_LIB       0
#define CPM_TYPE_GIB       1
        u8 type:1;
#define CPM_USER_RO        0
#define CPM_USER_RW        1
        u8 userw:1;
#define CPM_COMPOSITION_ADDITIVE          0
#define CPM_COMPOSITION_RESTRICTIVE       1
        u8 comp:1;
        s32 weight;
        u32 min;
        u32 max;
};
```

Listing A.1: The `cpm_awm` data structure

```
struct cpm_dev_dwr {
        struct device *dev;
        u8 id;
        char name[CPM_NAME_LEN];
        struct cpm_swm_range *swms;
        u8 swms_count;
        void *gov_data;
        void *pol_data;
        /* private */
        struct kobj_attribute kattr;
        struct attribute_group swms_group;

};
```

Listing A.2: The `cpm_dev_dwr` data structure

```
struct cpm_range {
        u32 lower;
        u32 upper;
#define CPM_SWM_TYPE_UNBOUNDED            0        /* no bounds defined
    */
#define CPM_SWM_TYPE_RANGE                1        /* upper and lover bound
    */
        /*  defined (if lower==upper */
        /*  than is a single value)  */
#define CPM_SWM_TYPE_LBOUND               2        /* lower bound only */
#define CPM_SWM_TYPE_UBOUND               3        /* upper bound only */
        u8 type:2;

};
struct cpm_swm_range {
        u8 id;
        struct cpm_range range;
        struct kobj_attribute kattr;
        char name[CPM_NAME_LEN];
};
```

Listing A.3: The `cpm_swm_range` data structure

```
struct cpm_fsc {
        u16 id;
        struct cpm_swm_range *swms;
        u8 swms_count;
        struct cpm_fsc_dwr *dwrs;
        u8 dwrs_count;
        void *gov_data;
        void *pol_data;
        struct list_head node;
};
```

Listing A.4: The `cpm_fsc` data structure

```
struct cpm_policy {
        char name[CPM_NAME_LEN];
        int (*sort_fsc_list) (struct list_head *fsc_list);
        int (*ddp_handler) (unsigned long event, void *data);
};
```

Listing A.5: The `cpm_policy` data structure

```
struct cpm_governor {
        char name[CPM_NAME_LEN];
        int (*build_fsc_list) (struct list_head *dev_list, u8 dev_count);
};
```

Listing A.6: The `cpm_governor` data structure

```
struct cpm_dev_data {
        ddp_callback notifier_callback;
        struct cpm_dev_dwr *dwrs;
        u8 dwrs_count;
};
```

Listing A.7: The `cpm_dev_data` data structure

The Policy type. Each policy is represented by a data structure `cpm_policy`, reported in Lis. A.5. When a policy module is loaded, its initialization code must setup an element of this type and passes it to the core. Along with a name attribute, the policy must define two callback:

- `int (*sort_fsc_list)(struct list_head *fsc_list);`
  is a reference to an asynchronous call that is used to notify the policy when a new sorted FSC list is requested. This methods takes the list of current FSC and computes asynchronously, to do not block the core, the new ordered list to notify back to the core.

- `int (*ddp_handler)(unsigned long event, void *data);`
  is used by the core after a constraint assertion to notify the policy during the FSC selection process. The first parameter is used to indicate the current step of the FSC selection, coded with the reported defines, according to the previously explained working flow. The second parameter is used instead differently. During the `CPM_EVENT_NEW_CONSTRAINT` event it is a reference to the newly asserted constraint. While, it points to the chosen FSC during all the other phases. The policy use this callback to interact with the core and assert if it is possible to continue with the FSC selection process.

The Governor type. Each governor is represented by a data structure `cpm_governor`, reported in Lis. A.6. When a governor module is loaded, its initialization code must setup an element of this type and passes it to the core. Along with a name attribute, the policy must define a callback:

- `int (*build_fsc_list)(struct list_head *dev_list, u8 dev_count);`
  is a reference to an asynchronous call that is used to ask the governor to build the FSC's list merging the DWRs of the devices that are currently registered to the framework. The DWR's can be found associated to the device's list which is passed with the first parameter. The second parameter simply define the number of devices in this list.core.

A.4.2 The frameworks' core

The core of the framework is defined in the `drivers/cpm/cpm_core.c` source file. The `cpm_core` entity has a fundamental role in the coordination of the framework activities and in managing communication between the other modules. To support these activities, a proper API is defined and exported by this entity. Let me review it starting from the calls related to the system initialization phase and then looking at the run-time support.

The initialization API. A set of functions are exported to support the system initialization phase. They are mainly related to entities registrations and data definitions. Lets me review the in details.

- `int cpm_register_platform_asms(struct cpm_platform_data *cpd);`
  is used by the platform code to register platform specific SWM. The input parameter `cpm_platform_data` essentially wraps a vector of `cpm_asm` shown in Lis. A.1, and a field that define their number. This function takes care of copying the SWMs into a core's local array. Moreover, each SWM is appended with a list of constraints that will be asserted on it. This allows to know at run-time the current QoS level which is granted by the system on each SWM.

- `int cpm_register_governor(struct cpm_governor *cg);`
  is used by governors to register to the core by passing a reference to a properly initialized structure of type `cpm_governor`, like the one shown in Lis. A.6.

- `int cpm_register_policy(struct cpm_policy *cp);`
  is used by policies to register to the core by passing a reference to a properly initialized structure of type `cpm_policy`, like the one shown in Lis. A.5.

- `int cpm_register_device(struct device *dev, struct cpm_dev_data *data);`
  is used by a device driver to register to the core by passing a reference to a properly initialized structure of type `cpm_dev_data`, like the one shown in Lis. A.7. This structure must be properly initialized by the device with an array containing its DWRs and a reference to the callback that must be used by the core to notify the device during a FSC selection process.

- `int cpm_set_fsc_list(struct list_head *fsc_list);`
  is used by a governor to return to the core the new FSCs that it has been built. Indeed, for performances reasons the FSC list is computed with an asynchronous process during which the core framework continues to run. When the governor has finished its computation it must notify the new FSCs list by passing it to the core using this function. After this call the core deallocates the old list and starts to use the new one.

- `int cpm_set_ordered_fsc_list(struct list_head *fscpl_head).`
  is used by a policy in a similar way of the previous one, to set up the new FSCs' ordered list. In this case also, the computation of this list is a non-blocking activity and so the policy must notify the new ordered list to the core using this function.

The run-time API. Two functions supports the constraint assertion and removal, accomplished by device drives at run-time. Lets me review them in details.

- `int cpm_update_constraint(struct device *dev, cpm_id asm_id, struct cpm_range * range);`
  allows a device driver to assert a new constraint, or to update one that it has previously asserted on the same SWM. To use this function, a driver must specify a pointer to itself, that will be used to track assertion's paternity, the ID of the desired SWM and a range that represent the level of QoS that it required.

  After this call, the core executes different actions. First of all it checks, searching in the list that it is appended to each SWM, if the considered device has already required a constraint on the same parameter. If this occurs, it substitute the existing with the new one. Then the core notifies for the first time the registered policy about the raised request and, if the policy grants its authorization, the core aggregates the required constraint to those that are currently asserted for that SWM. It can happen that after the aggregation the current FSC is no more valid: in this case the update process continues as explained in Par. A.3.2 on page 123

- `int cpm_remove_constraint(struct device *dev, cpm_id asm_id, struct cpm_range * range);`
  allows a device to communicate that it is no more interested in a previously requested level of QoS. The parameters that are needed for this action are similar to the assertion's case, with the only difference that the specified range must correspond to a previously requested one. After this request the core undoes the previous aggregation and, if this operation enable some previously invalidated FSCs a new distributed agreement process is triggered.

A.4.3 The user-space interface

The framework's sysfs interface is exported at this path `sys/kernel/cpm`. Under this directory, a file called `enable` allows to enable or disable CPM from the user space, by "echoing" respectively 1 or 0 into it. If CPM is disabled, devices, governors and policies can anyway register to the framework but, after that, CPM does not build any FSC list, does not manage any constraints assertion and does not search any FSC that corresponds to the optimal global system working point. In other words, if CPM is disabled, the system works in best-effort mode, without considering it. This possibility has been mainly provided for debugging purposes, but it can be useful also in normal circumstances. For instance, in order to avoid excessive overheads at boot-time, it is convenient to boot with the framework disabled.

Two folders can be found also at this path. They are used to export information on SWMs and FSCs.

The SWMs folder. It contains a file for each registered SWM. For an easy access, the file name correspond to the name given to the corresponding metrics. By simply reading one of these files we get some information on the metric formatted into the string:

```
ID:NAME TYPE COMPOSITION MIN MAX PERMISSION CONSTRAINT
```

where:

- `ID` is the numeric identifier of the SWM;
- `NAME` is the name of the SWM;
- `TYPE` is 'L' or 'G', depending whether the SWM is of type lower-is-better or greater-is-better;
- `COMPOSITION` is 'A' or 'R' depending if additive or restrictive composition is needed;
- `MIN` represents the minimum value allowed for the SWM;
- `MAX` represents the maximum value allowed for the SWM;
- `PERMISSION` can be 'w' if the SWM is user writable, '-' otherwise;
- `CONSTRAINT` shows the constraint currently asserted on the SWM. It can be 'UnB' if the SWM is unbounded, i.e. no constraints has been asserted.

This folder contains also two others files. The `constraint` file can be used to assert a new constraint on the SWM. This can be done by simply writing into it a string with this syntax:

```
ID:VALUE
```

where:

- `ID` is the ID of the SWM on which we want to assert a constraint;
- `VALUE` is a single numeric value that represents the constraint boundaries

If we are considering a LiB SWM this value will be interpreted as an upper bound otherwise, in a GiB case, as a lower bound.
The `weight` file allows instead to define a 'weight' for the SWM. This values can be used be an FSC ordering policy as a relative importance optimization value to each SWM. The usage of this file is similar to the previous one and required write a string with this syntax:

```
ID:WEIGHT
```

where:

- `ID` is the ID of the SWM on which we want to assign a weight;
- `VALUE` is a single numeric value that represents the weight.

The FSC folder. It contains a sub-folder for each identified FSC by the current governor. Inside each folder can be found a file for each SWM that define the corresponding FSC, which once read return a string with this syntax:

```
ID:NAME PERMISSION TYPE RANGE
```

where:

- `ID` is the ID of the corresponding SWM;
- `NAME` is the name of the SWM;
- `PERMISSION` can be 'w' if the SWM is user writable, '-' otherwise;
- `TYPE` indicates the kind of range is asserted by the FSC on the SWM. It can be 'R' in case of range, 'L' or 'U' for lower and upper bound respectively;
- `RANGE` shown instead the actual value of the range defined by the considered FSC on that SWM.

This directory contains also a link named `current` which points to the directory corresponding to the FSC that is active at that time.

A.5 Governor and Policies Examples

This section presents an example of policy and governor modules, which have been used to validate the framework and to run the tests described in Sec. 4.1 on page 99.

A.5.1 The performance policy

This policy has been implemented with the goal of testing the mechanism for the weighting of SWMs, used to implement the objective function. By using this policy it is possible to effectively support the power saving vs. performances optimization of the system. The two required callbacks are implemented that way:
The `sort_fsc_list` receives from the core the original list of FSCs and then scans all the elements contained inside it. For each FSC, it considers the set of SWMs that are involved in the FSC and retrieves the corresponding weight. These are used to evaluate the objective function for that SWM and then, by summing all of the obtained values, it evaluate the objective function for the FSC. At the end a new list of FSC ordered according to the computed results are registered to the core.
The `ddp_handler` is invoked by the core during the FSC selection process and, at each call, it always returns an agreement to the core. Indeed, this policy do not implement any access control.

A.5.2 The exhaustive governor

The *Exhaustive Governor* performs the identification of the FSCs which correspond to the set of DWRs declared by registered devices using an exhaustive search. This choice has allowed a simple and fast implementation, although this approach can surely be optimized to improve its efficiency. We have also chosen to provide a first implementation that exactly maps the definitions of DWRs and FSCs, as described in the theoretical model formulation, with the aim of proving the validity of the model itself. Moreover, an exhaustive research represents the intrinsic maximum computational complexity of the problem and thus it is also useful for stress testing purposes.

The `build_fsc_list` callbacks receives as input a reference to the list of devices that are currently registered to CPM. After some checks on the formal validity of the parameters the governor executes the algorithm that computes the FSCs. This search algorithm is based on a depth first on a graph data structure which is implemented using a recursive function. The pseudo-code of that function is represented in Lis. A.5. The recursive call has two exit conditions:

- the DWRs of the last device has been reached and all the SWMs' ranges merge with the 'candidate FSC' generating the new FSC;
- a range of the DWR currently considered does not merge with the related range of the 'candidate FSC'

This DWRs merge routine works by considering all the possible combinations of DWRs from all the devices. For each combination, it compares the first two DWRs, considering the common SWMs and computing the intersection of their ranges. If two such ranges have an empty intersection, then the two DWRs do not overlap. In this case, the comparison process is aborted and the algorithm continues with the next combination of DWRs. Otherwise, the result of a merge represents a 'candidate FSC'. Ranges expressed on SWMs which are not common to two DWRs are automatically selected to be part of the candidate FSC. Indeed, we can imagine to add to the DWR that do not consider this SWM an *unbounded range*, i.e. a range that space from the minimum to the maximum value declared on the metric.

When the comparison between the first two DWRs of a combination is completed, the same process is re-iterated considering the ranges of the candidate FSC, obtained at the previous step, and the next DWR of the combination. The process continue until all DWRs belonging to the combination have been considered, obtaining an FSC, or an empty merge has been found found, thus aborting the search and continuing the exploration.
The pseudo-code presented in Lis. A.6 shows the algorithm described above.

```
**Data**: Devices List, totalDevice
**Results**: list of found FSCs
$level \longleftarrow 1$;
$foundFSC \longleftarrow empty$;
recursive_fsc_search($device, level$);
begin
    forall DWR of device do
        forall ASM of DWR do
            $oldValue = findold(ASM, candRanges)$;
            if ($oldValue! = NULL$) then
                $newValue \longleftarrow merge(ASM, oldValue)$;
                if ($newValue! = NULL$) then
                    push($newValue, candRanges, level$);
                else
                    goto nomerge;
                end
            else
                push($ASM, candRanges, level$);
            end
        end
        if ($level == totalDevice$) then
            $newFSC \longleftarrow top(candRanges)$;
            add($newFSC, foundFSC$);
        else
            recursive_fsc_search($device- > next, level + 1$);
        end
        nomerge : pop(candRanges,level);
    end
end
```

.

Figure A.5: The pseudo-code of the FSC recursive search algorithm

```
Data: Devices' DWRs
Results: list of found FSCs
begin
    compute all the combinations composed by a DWR for each device;
    forall the computed combinations do
        CandidateFSC += merge ranges on a common SWM;
        if all merges are possible then
            CandidateFSC += ranges on SWMs not in common;
            IdentifiedFSC += CandidateFSC;
        else
            this combination do not generate and FSC;
        end
        CandidateFSC := empty;
    end
end
```

Figure A.6: The pseudo-code of the DWRs merge algorithm.

Appendix B

Main Linux Frameworks for Power Management

"If you want to accomplish something in the world, idealism is not enough – you need to choose a method that works to achieve the goal. In other words, you need to be pragmatic."

Richard Stallman

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. by directly operating on the supply voltage. In parallel, the higher software layer can be effectively employed in defining a suitable mechanism to provide the application with system-wide power management. There exist several software frameworks addressing power management. In this survey we focus on those designed for Linux-based systems, and which was originally designed for classical general purpose platforms, such as Intel processors for desktop computers. Nevertheless, their applicability is of (quite) general validity, also for mobile embedded systems.

This appendix review some of the main frameworks supporting power management that can be found already embedded into the mainline Linux kernel.

		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`)	●	●	?	?	?

Table B.1: 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, MVS, and power gating.

B.1 Frameworks Classification

Purpose of this section is to provide a concise overview of the different software support for power optimization. A summary of the available approaches is reported in Table B.1. We can identify two classes of software support: *i)* pure-OS and *ii)* cross-layer. The main differences rely on where the power optimization mechanisms are applied and which kind of interaction is exposed toward the user-space.

Pure-OS techniques are completely implemented within the Operating System; they do not provide support for direct input from software applications. They attempt to figure out application requirements, based on previously seen behavior or current activities, and enforce some control decision either on a single device or 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, also named *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.

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 centralized and distributed. *Centralized techniques* have been developed mainly to support the power optimization of relatively simple and dedicated embedded systems, for example personal media player, but 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, for example new generation smart-phones.

B.2 Pure-OS Techniques

We have to 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.

Being one of the more power demanding device, power optimization of the system processor is of major interest. Two are the 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 consumptions.

B.2.1 CPUidle: Do Noting Efficiently

This framework 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 semantics, there is no workload, and it can be turned out to avoid 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 wakeup latency. Moving from the simple approach of clock gating to power gating, there are increasing penalties, mostly related to wakeup 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.

The *CPUidle framework* [13] addresses power/performance trade-off from a software layer standpoint, with the target 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. To simplify the exploiting of such a sensitive power management techniques, the Linux implementation defines a proper software design which separate the low-level software mechanism from high level interface toward the framework clients. An overall view of the framework architecture is given in Fig. B.1.

The low-level interface 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 necessary to get 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 instead provides support for the definition of a *governor*,

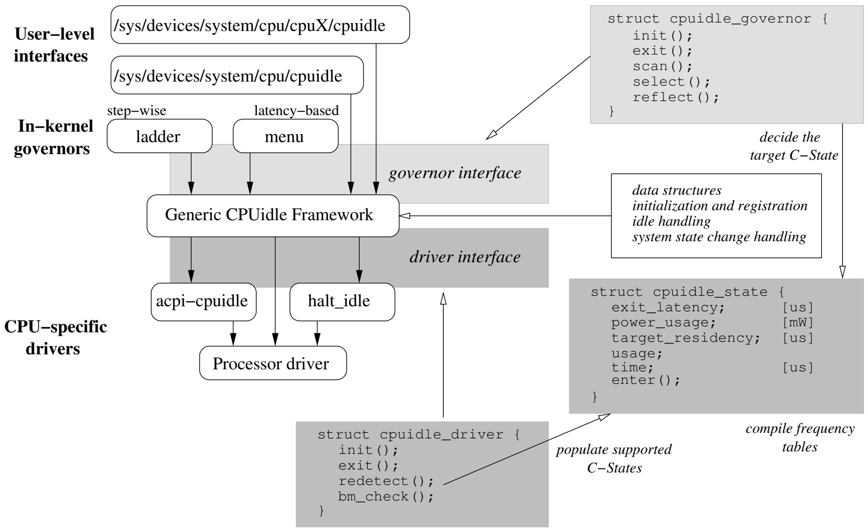

User–level interfaces
/sys/devices/system/cpu/cpuX/cpuidle
/sys/devices/system/cpu/cpuidle
In–kernel governors
step–wise
latency–based
ladder
menu
governor interface
Generic CPUidle Framework
driver interface
acpi–cpuidle
halt_idle
CPU–specific drivers
Processor driver
struct cpuidle_governor { init(); exit(); scan(); select(); reflect(); }
decide the target C–State
data structures
initialization and registration
idle handling
system state change handling
struct cpuidle_state { exit_latency; [us] power_usage; [mW] target_residency; [us] usage; time; [us] enter(); }
struct cpuidle_driver { init(); exit(); redetect(); bm_check(); }
populate supported C–States
compile frequency tables

Figure B.1: An overview of the CPUidle framework's software design.

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 and exploits the information on the maximum allowed system latency in order to better identify the idle state that should be reached every time there is the opportunity. This governor is certainly more efficient but requires a closer collaboration among applications and kernel drivers, to collect such requirements.

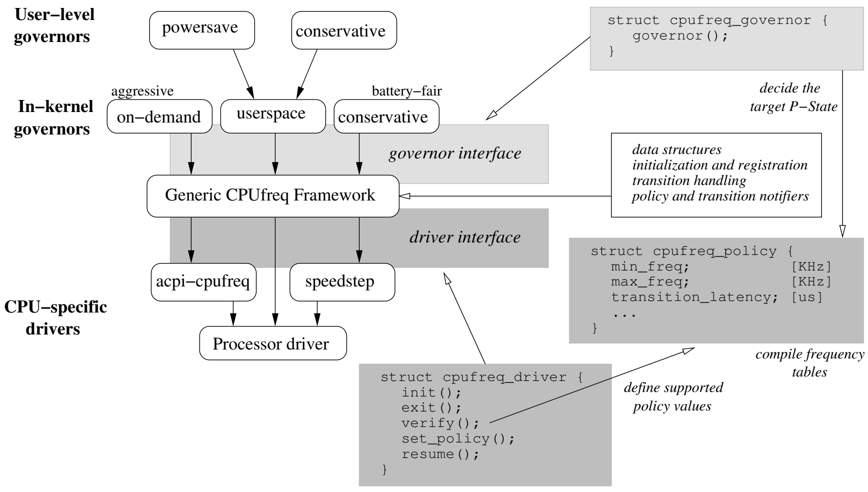
The core implementation is completely platform independent and provides the glue code that defines the required data structures, support drivers, governors registration and run-time selection. A proper monitoring interface is also exported to the collecting statistics on idle states usage.

B.2.2 CPUfreq: Use Just the Right Power

The *CPUfreq framework* [12] 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 performance states with reduced power consumption and increased execution time. Switching from a performance state to another inserts overhead that must be kept into consideration. Moreover, there is the need to efficiently identify the real system performance requirements. These observations make that of 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 Fig. B.2.

The low-level software mechanisms 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% [12]. On CPU utilization higher that that threshold, an immediate scaling up to the maximum available frequency is required, to the contrary, on



```
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();
}
```

Figure B.2: An overview of the CPUfreq framework's software design

lower CPU utilization the scaling down is required step-by-step but only after a pre-configured 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.

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. Indeed, these two components require system-broad optimization techniques which are able to collect information from multiple devices in order to identify a proper optimization strategy.

B.2.3 The Suspend/Resume Framework

The *Suspend/Resume Framework* 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 peripherals 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 interdependencies 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.

B.2.4 The 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 rationale on which the proposed framework is based is the management of the system clock signal. The hierarchical generation and distribution of the clock signal[1] as reported in Figure B.3 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. 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 those 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

[1]Usually, for performance reasons, through an H-tree

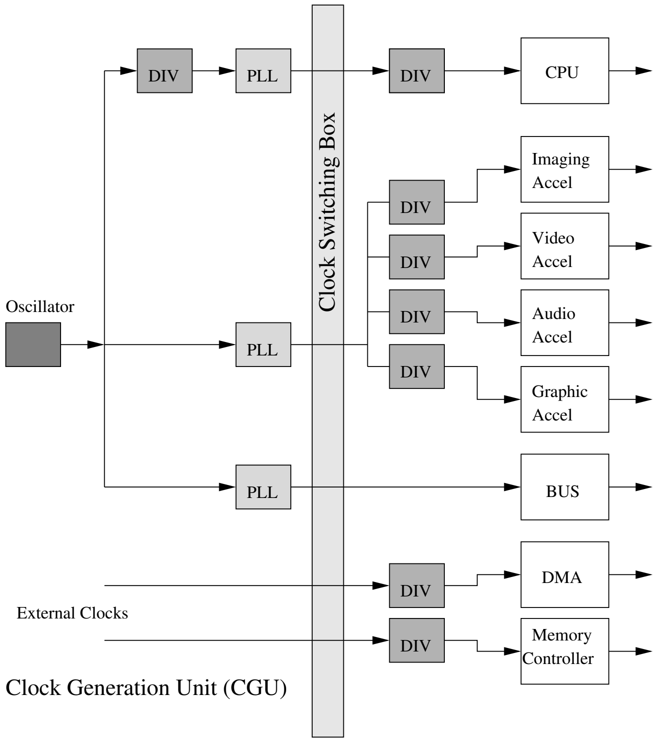


Figure B.3: An example of clock hierarchically.The clock signals are generated and distributed among the subsystems by means of several PLLs and DIV modules. This architecture of the clock distribution tree allows some level of control over the different system clocks.

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.

The software layer introduced by this framework completely and transparently hides the complexity of the clock generation and dependencies. The implementation provided by a modern Linux kernel is based on a minimal *abstract* interface to be implemented by the platform code of each machine. Every machine code that want use this framework is required to implement a set of callback routines which basically allows to: *a)* get a reference to a clock signal; *b)* set the required clock rate for a reference; *c)* release a reference once the clock is no more needed. This simple abstract API could mask a quite complex platform specific implementation. The platform code usually is in charge of defining a proper data structure to represent each clock signal available in the system and to track their hierarchical structure. Indeed, a simple call to a clock `set_rate()` method usually triggers a complex set of modifications that could also navigate all the clock tree up to PLLs in order to get out the expected result at the required clock end node.

By default, the abstract interface does not define a user-space interface, mainly for complexity and flexibility reasons, but generally every platform tends to export such an implementation for at least supporting debugging and monitoring. Unfortunately, the framework also does not keep track of inter-dependencies among devices, but only dependencies among Clock Generation Units (CGUs). Moreover, the framework cannot cover the entire platform. This is mainly due to the fact that some hardware resources, such as bus interconnects and memory L2 caches, do not have any associated device driver.

B.2.5 The Voltage and Current Control Framework

This framework provides a quite specific support focusing on the efficiency of voltage regulators. Modern SoC architectures are composed of multiple voltage domains to better fit specific requirements of each hardware block. In general, the voltage domains within a SoC could have some dependency relation between them and someone are directly controlled by a dedicated voltage regulator usually provided by an extern 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

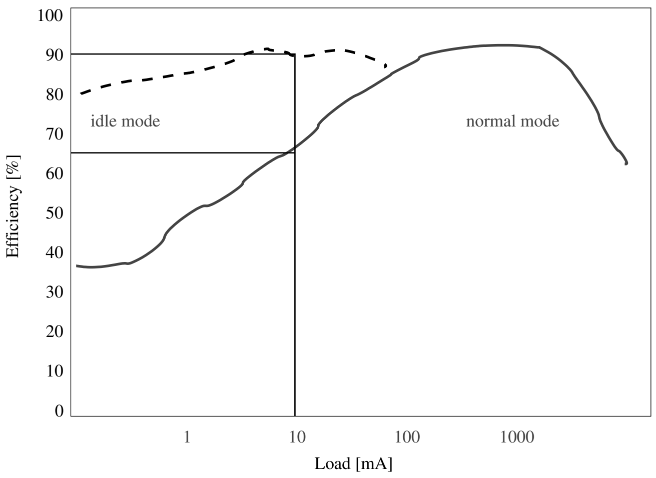


Figure B.4: The efficiency of a voltage regulator depends on the working mode and the current load.

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 B.1.

$$RPE = P_{out}/P_{in} \tag{B.1}$$

Equation B.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. A classical characterization of a regulator efficiency is very close to the graph depicted in Fig. B.4.

This diagram shows 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.

B.3 Cross-Layer Techniques

Mechanisms and techniques supporting power management can be implemented at different abstraction levels; not only at architectural and middleware abstraction levels but also at software level. Indeed, applications are aware of their Quality-of-Service 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.

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, implement only a lightweight, single, and system-wide optimization policy but exploit also 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.

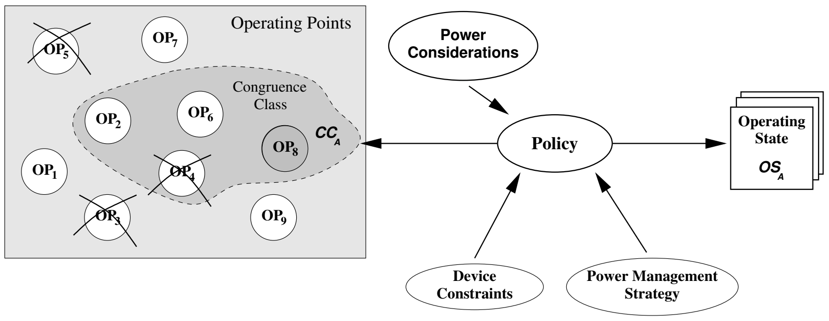


Figure B.5: The DPM architecture abstraction objects.

B.3.1 DPM: A centralized Approach to PM

The *Dynamic Power Management (DPM)* framework, presented in [97], 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 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 Fig. B.5.

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, 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 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, Indeed, according to the time 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 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 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.

B.3.2 QoSPM: A distributed Approach to PM

The *QoS 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.

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. A well defined and simple 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. [2]. 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, 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. Thus, the optimization policy implemented by CPUidle could easily discard all the idle states which have an exit time greater that the aggregated request on this parameter. 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 others specialized policies. It is worth noticing the current implementation of the notification mechanism support only a *best-effort* approach. Indeed, 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 any kind of feedbacks is 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. This design choice has allowed to easily export to the Linux kernel the paradigm of a cross-layer distributed approach to power management. 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.

[2]A notification chain is a standard in-kernel support to make asynchronous calls to a registered entity and are used to deliver events and data in a completely decoupled away.

Appendix C

Computational Complexity Analysis

"Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it. And to make matters worse: complexity sells better."

Edsger W. Dijkstra

This appendix present the complexity analysis of the main algorithms used by the proposed technique. First, a rapid review of computational complexity theory is given. This mostly aims at explaining the technical terms that are going to be used without any claim to be exhaustive considering also the theoretical complexity of the topic. Then, I analyze the algorithms with the main goal to focus the possible issues related to the amount of computational and temporal resources needed to execute the model.

C.1 Theory overview

Computational complexity theory [101] investigates the problems related to the resources required to execute algorithms. It is useful for analyse the scalability of a proposed solution and to place practical limits on what can and cannot be do with it. Computational complexity must be studied from a abstract standpoint, using only generic metrics that are technology independent. This last key point is absolutely mandatory, otherwise the same solution could have a different efficiency depending on the specific technology on which it is run.

This theory relies on a mathematical analysis of algorithms independently in the particular implementation and input data. The first step in the analysis is to abstract over the input and so, to find parameter(s) that characterize the *size* of the input, which is usually indicated by the variable n. The second step is to abstract over the implementation, in order to find a measure that pinpoints the running time of the algorithm not tied to a particular compiler or computer, for instance the number of arithmetic operations needed to complete the algorithm or the number of loops or the number of branches in a tree. Both the steps provide a measure of the number of steps taken by the algorithm as a function of the size of the input, which is indicated by the function $f(n)$.

The "Big O" notation. Complexity analysis is not trivial for two main reasons. First it hard to find a parameter n that completely give a characterization of the number of steps of an algorithm. Therefore, usually the worst-case and the average case are considered, computing $f_{WORST}(n)$ and $f_{AVG}(n)$. Secondly, it is hard to achieve an exact and precise analysis, therefore it is usually necessary to approximate. Hence the so called *Big O notation* is used. Briefly, "an algorithm is $O(n)$" means that its measure is at most a constant time n, with a possible exception of a few small values of n; formally:

$$f(n) = O(g(n)) \text{ if } \exists(n_0, c) \mid c \geq 0, n \geq 0, \forall n \geq n_0 \; f(n) \leq cg(n) \tag{C.1}$$

The function $f(n)$ must be monotonically increasing and computable in a finite time. The $O()$ notation defines the *asymptotic analysis*.

Computational Complexity Theory is based on a generic computation model, the Turing Machine, and on the measure of two generic metrics: *time* and *memory space* necessary to execute an algorithm.

Time complexity. This kind of complexity describes how the amount of time needed to solve a specific problem scales compared to the size of input data. More precisely we can say that a Turing Machine needs an amount of time $f(n)$ to complete a computation if, given an input of length n, it outputs the result after $f(n)$ elaboration's steps.

Space complexity. Analogously to the previous definition, this second kind of complexity describes how the amount of memory needed to solve a problem scales compared to the size of input data. We can say that a Turing Machine needs a memory space of size $f(n)$ if, given an input of length n it needs $f(n)$ temporary memory location to compute the results.

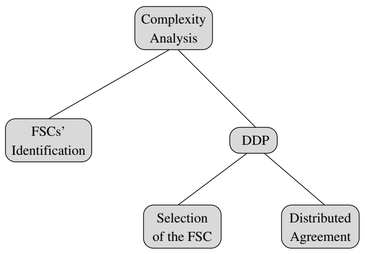


Figure C.1: Outline of the algorithms that will be presented and analyzed in terms of computational complexity

Optimum, Average and Worst case. Since it is not possible to evaluate precisely the time and the memory used to execute an algorithm, usually the optimum, average and worst case are considered.

- The *optimum* is the case where data are the best possible for the algorithm, i.e., those that require the less number of computations to be processed.
- The *average* is the most interesting case because provide an actual characterization of the complexity of the algorithm, but is also the hardest to be analyzed. It is often treated through repetitive simulations and an approximate value is statistically obtained.
- The *worst* is the case where data require the maximum number of steps and iterations of the algorithm.

To analyze the optimum case the Ω function is defined:

$$g(n) = \Omega(f(n)) \text{ if } \exists (n_0, c) \mid c \geq 0,\, n_0 \geq 0, \forall n \geq n_0 \; g(n) \leq cf(n) \qquad \text{(C.2)}$$

which indicates that $g(n)$ grows more slowly than $f(n)$: is an algorithm is $\Omega(n)$, it means that in the best case it requires $f(n)$ steps to be completed.

C.2 Technique analysis

In this analysis, I will focus on the three main steps of the model, outlined in Fig. C.1.

C.2.1 FSC identification

The 'FSC identification' step of the proposed technique, described in Sec. 3.4 on page 75, is comparable to the well-know Depth-first search over a well balanced tree. We report the detailed algorithm in pseudo-code.

- *currentDWR* is the DWR of a device currently analyzed by the algorithm.
- *candidateFSC* is the partial FSC brought on at each step of the algorithm and made of subsequent intersections of DWRs.
- *currentLevel* is the level of the tree currently analyzed.
- *foundFSCs* contains a list of found FSCs (partial results) and at the end of the algorithm the complete result.
- The *merge* function computes the intersection between a temporary FSC and the current DWR.
- The *insert* function adds a found FSC to the results.
- *Backtrack* moves the index of the algorithm a level up to the current one.

To analyze the computational complexity of the algorithm we define the following parameters:

- Branching factor (b): is the maximum number of successors of any node. In our case is the maximum number of $DWRs$ for a device.
- Maximum length (m) of any path considering the entire tree. In our case it is given by the number of devices + 1.

The worst case for the described algorithm corresponds to case where every path of the tree must be visited because every DWR results to have an intersection, or at most just the $DWRs$ of the last devices analyzed do not merge. The overall time complexity for the worst case is thus $O(b^m)$.
The space complexity in the worst case is given when an FSC is found for each path followed by the algorithm and result to be $O(bm + 1)$, then asymptotically $O(bm)$

The optimum case is when no DWR of the first device analyzed has an intersection with any DWR of any other device. Hence there is not any merge at all and the algorithm is blocked after having tried to intersect the DWR of the first device. The time complexity results to $\Omega(b)$.

```
Data: devices' DWRs
Results: list of identified FSCs
begin
    $currentDWR \longleftarrow treeRoot;$
    $candidateFSC[0] \longleftarrow unbounded;$
    $currentLevel \longleftarrow 0;$
    $foundFSCs \longleftarrow NULL;$
    forall paths in the tree do
        $currentDWR$ = next depth first search node;
        increase $currentLevel$ by 1;
        if (merge (currentDWR, candidateFSC[$currentLevel - 1$]) is not null)
        then
            $candidateFSC[currentLevel]$ = merge ($currentDWR$,
            $candidateFSC[currentLevel - 1]$;
            if (currentLevel is the last level of the tree) then
                insert $candidateFSC[currentLevel]$ in $foundFSCs$;
                backtrack;
                decrease $currentLevel$ by 1;
            end
        else
            backtrack;
            decrease $currentLevel$ by 1;
        end
    end
end
```

Figure C.2: FSC identification algorithm in pseudo-code.

C.2.2 FSC selection

The algorithm for the selection of a new *FSC* is not computationally expensive. It consists in scanning the elements of the set of identified *FSCs* and selecting the *first FSC* that matches the asserted constraint. The only variable that should be considered to evaluate the complexity is n: the number of *FSC* in the set.

The worst case is when the *entire* the list must be analyzed, eventually without finding a *FSC* that matches the constraint neither at the last step. I this case the time complexity is given by $O(n)$, and similarly even the space complexity results to be $O(n)$, i.e. the necessary memory for each *FSC*.

The optimum case is when the first element of the list matches the constraint and thus a *FSC* is soon carried out. The time and space complexity is $O(1)$ because just one element has to be analyzed.

C.2.3 Distributed agreement

Also the algorithm which drives the distributed agreement results trivial from the point of view of the complexity analysis and particularly it is linear with respect to the number of devices.

Hence, both time and space complexity are $O(m)$, where m is the number of devices. However, note that the computation complexity of the local policy of each device driver has not been considered because each driver can implement a different policy. If one wants to analyze specific cases the complexity of each driver's policy define a single contribute to be added linearly.

Finally the complexity of the complete Distributed Decision Process results to be the sum of two linear algorithms and thus it is linear too.

Bibliography

[1] G.E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, January 1998.

[2] R.H. Dennard, F.H. Gaensslen, V.L. Rideout, E. Bassous, and A.R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *Solid-State Circuits, IEEE Journal of*, 9(5):256–268, October 1974.

[3] G.G. Shahidi. Evolution of cmos technology at 32 nm and beyond. In *Custom Integrated Circuits Conference, 2007. CICC '07. IEEE*, pages 413–416, Sept. 2007.

[4] The International Technology Roadmap for Semiconductors. 2007 edition. Technical report, ITRS, 2007.

[5] David Culler, J.P. Singh, and Anoop Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1st edition, 1998. The Morgan Kaufmann Series in Computer Architecture and Design.

[6] W. Wolf, A.A. Jerraya, and G. Martin. Multiprocessor system-on-chip (mp-soc) technology. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(10):1701–1713, Oct. 2008.

[7] J.L. Manferdelli, N.K. Govindaraju, and C. Crall. Challenges and opportunities in many-core computing. *Proceedings of the IEEE*, 96(5):808–815, May 2008.

[8] F. Arakawa. Multicore soc for embedded systems. In *SoC Design Conference, 2008. ISOCC '08. International*, volume 01, pages I–180–I–183, Nov. 2008.

[9] Henry Kasim, Verdi March, Rita Zhang, and Simon See. Survey on parallel programming model. *Network and Parallel Computing*, pages 266–275, 2008.

[10] Takamichi Miyamoto, Saori Asaka, Hiroki Mikami, Masayoshi Mase, Yasutaka Wada, Hirofumi Nakano, Keiji Kimura, and Hironori Kasahara. Par-

allelization with automatic parallelizing compiler generating consumer electronics multicore api. *Parallel and Distributed Processing with Applications, International Symposium on*, 0:600–607, 2008.

[11] Simone Campanoni, Giovanni Agosta, and Stefano Crespi Reghizzi. A parallel dynamic compiler for cil bytecode. *SIGPLAN Not.*, 43(4):11–20, 2008.

[12] Venkatesh Pallipadi and Alexey Starikovskiy. The ondemand governor: past, present and future. In *Proceedings of Linux Symposium, vol. 2, pp. 223-238*, 2006.

[13] Venkatesh Pallipadi. cpuidle - do nothing, efficiently... In *Proceedings of the Linux Symposium*, 2007.

[14] Suresh Siddha, Venkatesh Pallipadi, and Arjan Van De Ven. Getting maximum mileage out of tickless. *Proceedings of the Linux Symposium*, 2007.

[15] Liam Girdwood. Every microamp is sacred - a dynamic voltage and current control interface for the linux kernel. Technical report, Wolfson Microelectronics, 2008.

[16] Antun Kras Boris Svilicic. Cmos technology: chalanges for future development, 2006.

[17] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 2–11, New York, NY, USA, 1996. ACM.

[18] Lawrence Spracklen and Santosh G. Abraham. Chip multithreading: Opportunities and challenges. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 248–252, Washington, DC, USA, 2005. IEEE Computer Society.

[19] Findlay Shearer. *Power Management in Mobile Devices*. Findlay Shearer, 2007.

[20] Vasanth Venkatachalam and Michael Franz. Power reduction techniques for microprocessor systems. *ACM Comput. Surv.*, 37(3):195–237, 2005.

[21] Massoud Pedram. Power optimization and management in embedded systems. In *ASP-DAC '01: Proceedings of the 2001 Asia and South Pacific Design Automation Conference*, pages 239–244, New York, NY, USA, 2001. ACM.

[22] Luca Benini and Giovanni de Micheli. System-level power optimization: techniques and tools. *ACM Trans. Des. Autom. Electron. Syst.*, 5(2):115–192, 2000.

[23] Michael Keating, David Flynn, Rob Aitken, Alan Gibbons, and Kaijian Shi. *Low Power Methodology Manual: For System-on-Chip Design*. Springer Publishing Company, Incorporated, 2007.

[24] Meeta Srivastav, S. S. S. P. Rao, and Himanshu Bhatnagar. Power reduction technique using multi-vt libraries. In *IWSOC '05: Proceedings of the Fifth International Workshop on System-on-Chip for Real-Time Applications*, pages 363–367, Washington, DC, USA, 2005. IEEE Computer Society.

[25] Hai Li, Swarup Bhunia, Yiran Chen, T. N. Vijaykumar, and Kaushik Roy. Deterministic clock gating for microprocessor power reduction. In *In Proc. of 9 th Int'l Symp. on High Performance Computer Architecture (HPCA*, pages 113–122, 2003.

[26] Hans Jacobson, Pradip Bose, Zhigang Hu, Alper Buyuktosunoglu, Victor Zyuban, Rick Eickemeyer, Lee Eisen, John Griswell, Doug Logan, Balaram Sinharoy, and Joel Tendler. Stretching the limits of clock-gating efficiency in server-class processors. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 238–242, Washington, DC, USA, 2005. IEEE Computer Society.

[27] Pokhrel Khem. Physical and silicon measures of low power clock gating success: An apple to apple case study. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. SNUG, 2007.

[28] Texas Instruments. *OMAP35xx Technical Reference Manual*, 2008.

[29] K. Joe, Hass David, and F. Cox. Level shifting interfaces for low voltage logic. In *9 th NASA Symposium on VLSI Design 2000*, page 4, 2000.

[30] Alexandru Andrei, Marcus T. Schmitz, Petru Eles, Zebo Peng, and Bashir M. Al Hashimi. Quasi-static voltage scaling for energy minimization with time constraints. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 514–519, Washington, DC, USA, 2005. IEEE Computer Society.

[31] Saewong Saowanee and Rajkumar Ragunathan Raj. Optimal static voltage-scaling for real-time systems, 2001.

[32] Kihwan Choi, Wonbok Lee, Ramakrishna Soma, and Massoud Pedram. Dynamic voltage and frequency scaling under a precise energy model considering variable and fixed components of the system power dissipation. In *IEEE/ACM International Conference on Computer Aided Design*, November 2004.

[33] Ravindra Jejurikar, Cristiano Pereira, and Rajesh Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *DAC '04: Proceedings of the 41st annual conference on Design automation*, pages 275–280, New York, NY, USA, 2004. ACM.

[34] Sandeep Dhar, Dragan Maksimović, and Bruno Kranzen. Closed-loop adaptive voltage scaling controller for standard-cell asics. In *ISLPED '02: Proceedings of the 2002 international symposium on Low power electronics and design*, pages 103–107, New York, NY, USA, 2002. ACM.

[35] Roy Liu H. Multi-voltage adaptive voltage scaling soc reference design. Technical report, National Semiconductor Corporation, 2006.

[36] Kanak Agarwal, Kevin Nowka, Harmander Deogun, and Dennis Sylvester. Power gating with multiple sleep modes. In *ISQED '06: Proceedings of the 7th International Symposium on Quality Electronic Design*, pages 633–637, Washington, DC, USA, 2006. IEEE Computer Society.

[37] A. Sathanur, A. Pullini, L. Benini, A. Macii, E. Macii, and M. Poncino. A scalable algorithmic framework for row-based power-gating. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 379–384, New York, NY, USA, 2008. ACM.

[38] Compaq Computer Corporation and Revision B. Advanced configuration and power interface specification, 2000.

[39] Anna R. Karlin, Mark S. Manasse, Lyle A. McGeoch, and Susan Owicki. Competitive randomized algorithms for non-uniform problems. In *SODA '90: Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 301–309, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[40] Canturk Isci, Alper Buyuktosunoglu, and Margaret Martonosi. Long-term workload phases: Duration predictions and applications to dvfs. *IEEE Micro*, 25(5):39–51, 2005.

[41] Alan M. Turing. Computability and lambda-definability. *J. Symb. Log.*, 2(4):153–163, 1937.

[42] Kelvin D. and Bernt Rygg. Worst-case execution time analysis on modern processors. *SIGPLAN Not.*, 30(11):20–30, 1995.

[43] Sung-Soo Lim, Young Hyun Bae, Gyu Tae Jang, Byung-Do Rhee, Sang Lyul Min, Chang Yun Park, Heonshik Shin, Kunsoo Park, Soo-Mook Moon, and Chong Sang Kim. An accurate worst case timing analysis for risc processors. *IEEE Trans. Softw. Eng.*, 21(7):593–604, 1995.

[44] Pedro Diniz. A compiler approach to performance prediction using empirical-based modeling. *Computational Science – ICCS 2003*, pages 698–698, 2003.

[45] T. Ishihara and H. Yasuura. Voltage scheduling problem for dynamically variable voltage processors. In *Low Power Electronics and Design, 1998. Proceedings. 1998 International Symposium on*, pages 197–202, August 1998.

[46] Xiaobo Fan, Carla S. Ellis, and Alvin R. Lebeck. The synergy between power-aware memory systems and processor voltage scaling. In *In Workshop on Power-Aware Computing Systems*, pages 164–179, 2003.

[47] Thomas L. Martin and Daniel P. Siewiorek. Nonideal battery and main memory effects on cpu speed-setting for low power. *IEEE Trans. Very Large Scale Integr. Syst.*, 9(1):29–34, 2001.

[48] Giorgio C. Buttazzo. Scalable applications for energy-aware processors. In *EMSOFT '02: Proceedings of the Second International Conference on Embedded Software*, pages 153–165, London, UK, 2002. Springer-Verlag.

[49] M. Fleischmann. Longrun power management - dynamic power management for crusoe processors. Technical report, Transmeta Corp., 2001.

[50] *Crusoe TM5600 Processor Data Sheet*, 2006.

[51] D. Genossar and N. Shamir. Intel pentium m power estimation, budgeting, optimization, and validation. Technical report, Intel Corporation, May 2003.

[52] Intel Corporation. Wireless intel speedstep power manager, 2004.

[53] ARM Corp. Intelligent energy manager (iem) hardware control system in the arm1176jzf-s development chip. Technical report, ARM Corp., 2006.

[54] Suji Velupillai and Ken Tough. Intelligent energy manager (iem) benchmarking on a freescale's imx31 multimedia processor. Technical report, ARM Corp., 2006.

[55] Qiang Wu, Philo Juang, Margaret Martonosi, and Douglas W. Clark. Formal online methods for voltage/frequency control in multiple clock domain microprocessors. *SIGARCH Comput. Archit. News*, 32(5):248–259, 2004.

[56] Mark Weiser, Brent Welch, Alan Demers, and Scott Shenker. Scheduling for reduced cpu energy. In *OSDI '94: Proceedings of the 1st USENIX conference on Operating Systems Design and Implementation*, page 2, Berkeley, CA, USA, 1994. USENIX Association.

[57] Kinshuk Govil, Edwin Chan, and Hal Wasserman. Comparing algorithm for dynamic speed-setting of a low-power cpu. In *MobiCom '95: Proceedings of the 1st annual international conference on Mobile computing and networking*, pages 13–25, New York, NY, USA, 1995. ACM.

[58] Krisztián Flautner and Trevor Mudge. Vertigo: automatic performance-setting for linux. *SIGOPS Oper. Syst. Rev.*, 36(SI):105–116, 2002.

[59] Krisztián Flautner, Steve Reinhardt, and Trevor Mudge. Automatic performance setting for dynamic voltage scaling. In *MobiCom '01: Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 260–271, New York, NY, USA, 2001. ACM.

[60] Jacob R. Lorch and Alan Jay Smith. Improving dynamic voltage scaling algorithms with pace. In *SIGMETRICS '01: Proceedings of the 2001 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 50–61, New York, NY, USA, 2001. ACM.

[61] Jacob R. Lorch and Alan Jay Smith. Operating system modifications for task-based speed and voltage. In *MobiSys '03: Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 215–229, New York, NY, USA, 2003. ACM.

[62] A. E. Papathanasiou and M. L. Scott. Increasing disk burstiness for energy efficiency. Technical report, University of Rochester, Rochester, NY, USA, 2002.

[63] Sudhanva Gurumurthi, Anand Sivasubramaniam, Mahmut Kandemir, and Hubertus Franke. Drpm: dynamic speed control for power management in server class disks. *SIGARCH Comput. Archit. News*, 31(2):169–181, 2003.

[64] Robin Kravets and P. Krishnan. Power management techniques for mobile communication. In *MobiCom '98: Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, pages 157–168, New York, NY, USA, 1998. ACM.

[65] N. Pettis and Yung-Hsiang Lu. A homogeneous architecture for power policy integration in operating systems. *Computers, IEEE Transactions on*, 58(7):945–955, July 2009.

[66] C.S. Ellis. The case for higher-level power management. In *Hot Topics in Operating Systems, 1999. Proceedings of the Seventh Workshop on*, pages 162–167, 1999.

[67] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. Ecosystem: managing energy as a first class operating system resource. *SIGPLAN Not.*, 37(10):123–132, 2002.

[68] Heng Zeng, Carla S. Ellis, Alvin R. Lebeck, and Amin Vahdat. Currentcy: a unifying abstraction for expressing energy management policies. In *ATEC '03: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 4–4, Berkeley, CA, USA, 2003. USENIX Association.

[69] Cristiano Pereira, Rajesh Gupta, and Mani Srivastava. Pasa: A software architecture for building power aware embedded systems. In *In Proceedings of the IEEE CAS Workshop on Wireless Communication and Networking*, 2002.

[70] Manish Anand, Edmund B. Nightingale, and Jason Flinn. Ghosts in the machine: interfaces for better power management. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 23–35, New York, NY, USA, 2004. ACM.

[71] Taliver Heath, Eduardo Pinheiro, Jerry Hom, Ulrich Kremer, and Ricardo Bianchini. Code transformations for energy-efficient device management. *IEEE Transactions on Computers*, 53(8):974–987, 2004.

[72] Shivajit Mohapatra, Radu Cornea, Nikil Dutt, Alex Nicolau, and Nalini Venkatasubramanian. Integrated power management for video streaming to mobile handheld devices. In *MULTIMEDIA '03: Proceedings of the eleventh ACM international conference on Multimedia*, pages 582–591, New York, NY, USA, 2003. ACM.

[73] J. Pouwelse, K. Langendoen, and H.J. Sips. Application-directed voltage scaling. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 11(5):812–826, Oct. 2003.

[74] D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: an operating system architecture for application-level resource management. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 251–266, New York, NY, USA, 1995. ACM.

[75] Gregory R. Ganger, Dawson R. Engler, M. Frans Kaashoek, Hector M. Brice Russell Hunt, and Thomas Pinckney. Fast and flexible application-level networking on exokernel systems. *ACM Trans. Comput. Syst.*, 20(1):49–83, 2002.

[76] T. K. Tan, A. Raghunathan, and N. K. Jha. Software architectural transformations: A new approach to low energy embedded software. In *DATE '03: Proceedings of the Conference on Design, Automation and Test in Europe*, page 11046, Washington, DC, USA, 2003. IEEE Computer Society.

[77] Malena Mesarina and Yoshio Turner. Reduced energy decoding of mpeg streams. *Multimedia Syst.*, 9(2):202–213, 2003.

[78] Donghwan Son, Chansu Yu, and Heung-Nam Kim. Dynamic voltage scaling on mpeg decoding. In *Parallel and Distributed Systems, 2001. ICPADS 2001. Proceedings. Eighth International Conference on*, pages 633–640, 2001.

[79] Kihwan Choi, Karthik Dantu, Wei-Chung Cheng, and Massoud Pedram. Frame-based dynamic voltage and frequency scaling for a mpeg decoder. In *ICCAD '02: Proceedings of the 2002 IEEE/ACM international conference on Computer-aided design*, pages 732–737, New York, NY, USA, 2002. ACM.

[80] Johan Pouwelse, Koen Langendoen, Inald Lagendijk, and Henk Sips. Power-aware video decoding. In *in 22nd Picture Coding Symposium, Seoul, Korea*, pages 303–306, 2001.

[81] Zhijian Lu, Jason Hein, Marty Humphrey, Mircea Stan, John Lach, and Kevin Skadron. Control-theoretic dynamic frequency and voltage scaling for multimedia workloads. In *CASES '02: Proceedings of the 2002 international conference*

on Compilers, architecture, and synthesis for embedded systems, pages 156–163, New York, NY, USA, 2002. ACM.

[82] Kihwan Choi, K. Dantu, Wei-Chung Cheng, and M. Pedram. Frame-based dynamic voltage and frequency scaling for a mpeg decoder. In *Computer Aided Design, 2002. ICCAD 2002. IEEE/ACM International Conference on*, pages 732–737, Nov. 2002.

[83] Xiaotao Liu, Prashant Shenoy, and Mark D. Corner. Chameleon: Application-level power management. *IEEE Transactions on Mobile Computing*, 7(8):995–1010, 2008.

[84] D.G. Sachs, S.V. Adve, and D.L. Jones. Cross-layer adaptive video coding to reduce energy on general-purpose processors. In *Image Processing, 2003. ICIP 2003. Proceedings. 2003 International Conference on*, volume 3, pages III–109–12 vol.2, Sept. 2003.

[85] Prashant J. Shenoy and Peter Radkov. Proxy-assisted power-friendly streaming to mobile devices. *Multimedia Computing and Networking 2003*, 5019(1):177–191, 2003.

[86] Morihiko Tamai, Tao Sun, Keiichi Yasumoto, Naoki Shibata, and Minoru Ito. Energy-aware video streaming with qos control for portable computing devices. In *NOSSDAV '04: Proceedings of the 14th international workshop on Network and operating systems support for digital audio and video*, pages 68–73, New York, NY, USA, 2004. ACM.

[87] Germany. Proceedings of Middleware 2001: IFIP/ACM International Conference on Distributed Systems Platforms, editor. *Reducing the Energy Usage of Office Applications*, 2001.

[88] Eyal De Lara, Dan S. Wallach, and Willy Zwaenepoel. Puppeteer: Component-based adaptation for mobile computing. In *USITS'01: Proceedings of the 3rd conference on USENIX Symposium on Internet Technologies and Systems*, pages 14–14, Berkeley, CA, USA, 2001. USENIX Association.

[89] Brian Noble, M. Satyanarayanan, and Morgan Price. A programming interface for application-aware adaptation in mobile computing. In *MLICS '95: Proceedings of the 2nd Symposium on Mobile and Location-Independent Computing*, pages 57–66, Berkeley, CA, USA, 1995. USENIX Association.

[90] Jason Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *SOSP '99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, pages 48–63, New York, NY, USA, 1999. ACM.

[91] Angela B. Dalton and Carla S. Ellis. Sensing user intention and context for energy management. In *HOTOS'03: Proceedings of the 9th conference on Hot*

Topics in Operating Systems, pages 26–26, Berkeley, CA, USA, 2003. USENIX Association.

[92] Wanghong Yuan and Klara Nahrstedt. Energy-efficient soft real-time cpu scheduling for mobile multimedia systems. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 149–163, New York, NY, USA, 2003. ACM.

[93] Yunsi Fei, Lin Zhong, and Niraj K. Jha. An energy-aware framework for coordinated dynamic software management in mobile computers. In *MASCOTS '04: Proceedings of the The IEEE Computer Society's 12th Annual International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunications Systems*, pages 306–317, Washington, DC, USA, 2004. IEEE Computer Society.

[94] Nevine AbouGhazaleh, Bruce Childers, Daniel Mosse, Rami Melhem, and Matthew Craven. Energy management for real-time embedded applications with compiler support. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 284–293, New York, NY, USA, 2003. ACM.

[95] Jerry Hom and Ulrich Kremer. Energy management of virtual memory on diskless devices. *Compilers and operating systems for low power*, pages 95–113, 2003.

[96] Joris S. M. Vergeest, Wolf Y. Song, and Han J. Broek. On the synthesis of discrete controllers. In *Proc. Tools and Methods of Competitive Engineering*, pages 731–739, 1995.

[97] Bishop Brock and Karthick Rajamani. Dynamic power management for embedded systems. *Proceedings on IEEE International SoC Conference*, pages 416–419, Sept. 2003.

[98] Jirí Matouek and Bernd Gärtner. *Understanding and Using Linear Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[99] STMicroelectronics. *Nomadik STn8815 Reference Manual*, 2007.

[100] Mark Gross. Pm quality of service interface. Technical report, Linux Kernel Documentation, 2008.

[101] Ding-Zhu Du and Ker-I Ko. *Theory of Computational Complexity*. Wiley-Interscience, 2000.

I am not to blame for putting forward, in the course of my work on science, any general rule derived from a previous conclusion.

Leonardo Da Vinci

Politecnico di Milano
Dipartimento di Elettronica e Informazione
Piazza Leonardo da Vinci, 32 – 20133 MILANO