Power Management for Portable Devices

J.A. Pouwelse

Power Management for Portable Devices

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.dr.ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op maandag 20 oktober 2003 om 13:00 uur

door

Janis Adriaan POUWELSE

informatica ingenieur,

geboren te Middelburg.

Dit proefschrift is goedgekeurd door de promotor:
Prof.dr.ir. H.J. Sips

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof.dr.ir. H.J. Sips,	Technische Universiteit Delft, promotor
Prof.dr.ir. R.L. Lagendijk,	Technische Universiteit Delft
Prof.dr. L. Benini,	University of Bologna, Italy
Prof.dr.ir. E. Deprettere,	Universiteit Leiden
Prof.dr.ir. T. Krol,	Universiteit Twente
Dr.ir. J.F.C.M. de Jongh,	TNO-FEL, Den Haag
Dr. K.G. Langendoen,	Technische Universiteit Delft

Dr. K.G. Langendoen heeft als begeleider in belangrijke mate aan de totstandkoming van het proefschrift bijgedragen.

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Pouwelse, Janis Adriaan

Power Management for Portable Devices
Janis Adriaan Pouwelse. -
Delft: Delft University of Technology
Thesis Technische Universiteit Delft. - With index, ref. - With summary in Dutch
ISBN 90-6464-993-6
NUGI 850
Subject headings: power management; portable devices; ACPI; low-power; energy efficiency

Correspondence: j@mp3.nl

Printed in the Netherlands

Dedicated to lost family

Contents

Chapter 1

Introduction

PEOPLE in the western world use many electronic devices in their daily lives. Due to significant technological advances, these devices have ever increasing capabilities. Various *portable* electronic devices have emerged that are small enough to fit on a wrist or to be carried in a briefcase. The research described in this thesis is about such portable devices. Portable devices usually operate on batteries with little capacity. The central research question in this thesis is how to increase the efficiency of battery usage so that battery lifetime can be measured in days instead of hours. This first chapter gives an introduction to portable devices and to the trends in their technological development.

Roadmap

1.1 Portable devices

The number of portable devices sold each year is increasing rapidly. Especially cell phones are sold by the millions. World-wide sales for 2002 are estimated at 430 million phones [140]. The mobile-phone industry is currently the largest consumer electronics segment in the world. A large percentage of the people in the western world is hooked on their cell phone and use it daily. The cell phone is bucking to replace the personal computer as the most ubiquitous piece of technology in our lives. The universal need for communication is the driving force behind the success of cell phones. Other popular forms of communication via portable devices, besides talking, are email, chatting, and paging.

Figure 1.1. A combination of a PDA and GSM phone sold by Handspring.

Another growing application area for portable devices is entertainment. Cell phones are increasingly becoming more than just devices for voice communication as they have custom dial tones, different screen logos, and interchangeable covers to change the color of your phone. Playing music from a portable device is another popular entertainment application. The FM radio, Mini-Disc player, and portable audio CD player are found in every electronics store. As of 2001, MP3 players are no longer targeted towards a special audience, but are offered as a standard consumer electronics product. Portable music has improved significantly since the days of the cassette tape; a collection with 500 hours of music now fits inside a shirt pocket. Gaming devices provide portable entertainment. Portable gaming devices were pioneered by Nintendo with the Gameboy. The gaming devices evolved from simple toys into powerful computers with the progress of technology. These gaming devices turn kids into young consumers that think digitally and prepare them for the laptop or Personal Digital Assistant (PDA) market.

Information access, manipulation, and processing is also a significant market for portable devices. Instead of using a paper agenda, people are turning to a PDA for organizing their lives. Analog cameras are being replaced with digital counterparts. Laptop computers are dropping in price and have become affordable for a large audience. In business, the laptop computer is common. Laptops with wireless connections to the Internet are now appearing, enabling full user mobility. Access to the World Wide Web (WWW) from a laptop offers the user a rich set of services. Information access from portable devices, however, is still in its infancy. The world leader is the Japanese telecom operator NTT Docomo, which started its I-mode service only as recent as 1999 [137].

A significant trend in portable devices is to reduce size. For several functions, devices have been shrunk towards their ultimate size: the size of an ordinary wrist watch. An even smaller size would further compromise usability. Wrist watch models now exist on the market for GSM, GPS location, photo camera, and MP3 audio applications.

An increasing number of devices become more versatile, programmable, and flexible. For example, some mobile phones can also play music and take pictures. Instead of being limited to a single

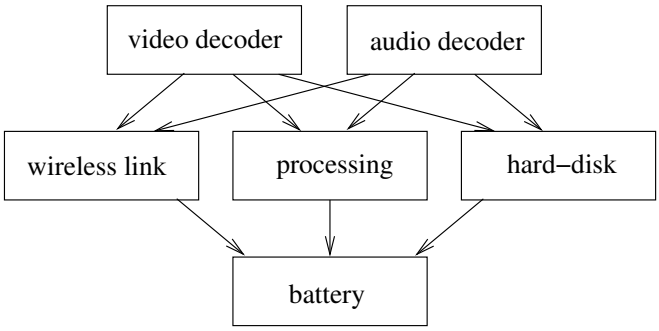


Figure 1.2. The components of the wireless multimedia playback device used throughout this thesis.

application, the user can extend the device with hardware and/or software. PDAs such as the Palm Pilot offer hardware slots for expanding their capabilities, similar to the larger PCMCIA slots of laptops. In 2001 the first Java phones were marketed in Japan, enabling downloading and execution of new software. This versatility also stimulates the shift towards *multi-modality*. Multi-modality means that the user interface supports multiple methods of interaction with the device, such as speech, motion, or even gestures for portable devices equipped with a camera. Portable devices for information are no longer limited to plain text and devices for communication provide more than mere voice services. Pictures and movies are becoming increasingly important in the mobile world, indicating a trend towards portable multimedia devices.

The future of portable devices is difficult to predict. The blurring distinction between cell phones, Gameboys, and PDAs is taken by some people as evidence that within several years these different functions will be merged into a single device. Within this future scenario, the dominant type of device will be the generic multi-purpose device that can be used for a wide range of applications. An example of a mix between a PDA and cell phone is shown in Figure 1.1. This device is equipped with a 33 MHz general-purpose processor, uses the PalmOS, and can connect to a GSM network [75]. An alternative scenario is that single-purpose devices will prevail. A single-purpose design means very competitive products due to high levels of integration and optimization. Consequently, such single-purpose devices are not programmable. Within this scenario, multi-purpose devices such as depicted in Figure 1.1 will not be popular. Single-purpose devices are specialized for either 'hand-held usage' or 'device-to-ear usage', and therefore easier to use. Another possible future scenario is that both multi-purpose and single-purpose devices will co-exist.

There are severe problems with portable devices. The cell phone has a limited talk time and may die in the middle of a conversation. The portable MP3 music player can store a collection of 500 hours, but the batteries last less than 24 hours. The Nintendo Gameboy has a display that is very difficult to read. A superior Gameboy display would result in an unreasonably short battery lifetime. A laptop only works for a few hours; after that it just becomes an unusable brick. The GPS locator that fits on your wrist can only measure your position continuously for 4 hours on one battery.

The above examples illustrate that battery lifetime is an overall problem for portable devices. Within this thesis we specifically target the problem of power management to prolong the battery lifetime of portable devices. We focus on the power management of portable devices with a general-purpose microprocessor and general-purpose OS (multi-purpose scenario). Of particular interest are

portable devices with both a wireless link and multimedia playback capability. The components of such a wireless multimedia device are shown in Figure 1.2. Such wireless multimedia devices are interesting because consumers demand them and companies have so far been unable to supply them. The wireless multimedia devices that were created up to 2003 are still not up to the task: more research and development is needed. Common problems with such devices are their insufficient performance, large size, high price, and limited battery lifetime.

1.2 Power trends

Power consumption is *the limiting factor* for the functionality offered by portable devices that operate on batteries [55].

This power consumption problem is caused by a number of factors. Users are demanding more functionality, more processing, longer battery lifetimes, and smaller size, but hardware prices must roughly remain the same. Battery technology is only progressing slowly; the performance improves just a few percent each year. Portable devices are also getting smaller and smaller, implying that the amount of space for batteries is also decreasing. Decreasing size results in less energy storage and a need for less power consumption. Users do not accept a battery lifetime of less than an hour; for wrist-watch-like devices even lifetimes of several months are expected. New, more powerful processors appear on the market that can deliver the performance users desire for their new applications. Unfortunately, these powerful processors often have a higher power consumption than their predecessors. Finally, users do not only want more processing, but also more features such as multimedia, mass storage, always-on wireless access, and speech recognition.

It is important to utilize the available energy within batteries as efficiently as possible to meet user demands. Energy preservation, or energy management, is further translated into a low power consumption by all parts of a portable device. Sophisticated power management is an important requirement to increase the battery life, usability, and functionality of devices.

Power consumption is the rate at which energy is consumed. With the fixed energy capacity of a battery, the power consumption directly determines the battery lifetime of a portable device. Throughout this thesis the reader is assumed to be familiar with the difference between power and energy. The challenge is doing as much as possible with the lowest amount of energy. Efficiency is the key to solving the power crisis. High performance with high power consumption does not necessarily mean less energy efficient and conversely, low performance and low power consumption does not mean that a device is more energy efficient [111]. For example, a component may consume five times more power and deliver ten times the performance of alternatives. Such a component would double the efficiency. The primary concern throughout this thesis is improving the energy efficiency, even if it requires that we temporarily increase power consumption. Note that we use the term *component* within this thesis for both hardware and software entities, such as a processor, hard disk, video decoder, and web browser.

The power consumption of portable devices is not dominated by a single component. Several studies have investigated the power consumption of portable devices [87; 119; 178; 182]. The main conclusion is that there is no single component or single activity that dominates the power consumption in a portable device. Therefore, the power consumption of *all* components needs to be reduced to lower the total amount of power. This section briefly discusses the problems and future technology directions for a number of components of portable devices. These components form the basis of wireless multimedia playback devices as shown in Figure 1.2.

1.2.1 Power supplies

Portable power generation and storage is a field that has seen relatively little technology progress over the last years. Batteries are currently the dominating technology for powering portable devices. The International Roadmap for Semiconductors [3] predicts that the average power that can be drained from a battery within a hand-held device will increase from 2.0 W in 2001 towards 2.3 W in 2011.

Several battery technologies exist. Laptops are generally equipped with lithium-ion batteries that contain no metallic lithium, as opposed to lithium-polymer batteries. Lithium-ion batteries in a laptop with a weight of 100 g deliver around 10 Wh, heavily depending on the shape. For small-sized PDA devices, non-rechargeable batteries can be used. A standard AAA sized alkaline battery has a weight of less than 12 g and a performance of around 1.4 Wh at a cost of approximately $1.

Alternative sources exist to provide the power for a wearable device. Amongst the alternatives, fuel cells are the most promising new technology. Fuel cells give a significant improvement over traditional batteries. Several types of fuel cells exist, such as the alkaline fuel cells that have been used by NASA on space missions, methanol fuel cells that can be easily re-filled, proton exchange membrane fuel cells that use hydrogen, and the exotic air-breathing aluminum-air fuel cells. Aluminum-air fuel cells are claimed to potentially offer 75 times more energy density than lithium-ion batteries [132]. Laboratory implementations yielding 800 Wh/kg have been achieved. Commercial products, that is, small and affordable fuel cells, are predicted to be a few years away. It is unclear if end users are willing to spend a few dollars when their methanol, hydrogen, or aluminum cartridges are empty. A factor against hydrogen fuel cells is their explosion potential. For instance, it remains to be seen if hydrogen-powered portable devices will be allowed on airplanes.

Solar-powered devices are also far away from mainstream use due to practical limits on sunlight exposure and low efficiency. In [176] an extensive study is conducted on the technique to drain power from the user itself. Wearing a neck brace that converts body heat into energy yields 0.16 W at 50 % efficiency, insufficient for the majority of applications. Several prototypes have been made of piezoelectric shoe inserts that generate power while the wearer is walking. The prototype described in [108] generates just a fraction of a Watt.

A conclusion from the above is that power supplies are not likely to deliver more performance within the coming years, hence we must find a way to do more with the limited energy we have.

1.2.2 Processing

The generic term *processing* covers the whole range of hardware components that manipulate data, ranging from dedicated hardware to general-purpose microprocessors. An example of dedicated hardware is the processing of the wireless signals within a cell phone. A PDA equipped with a microprocessor offers much more flexibility for processing and supports multiple applications.

Dedicated hardware using current technology obtains an efficiency of 0.5 pJ per operation (pJ/op), general-purpose microprocessors use *three* orders of magnitude more power, 500 pJ/op for the same operation [28]. Therefore, the flexibility that PDAs offer comes at a very high price.

Flexibility is very expensive in terms of power efficiency, but it is still demanded. For example, users demand support for the latest multimedia services and standards. Power-efficient dedicated hardware cannot be modified by the end user, leaving the inefficient microprocessor as the only viable option in most applications. Within this thesis we assume that the end user of a portable device has diverse needs that change over time and require the flexibility that only a microprocessor can offer.

The power dissipation of general-purpose processors has been rapidly increasing along with increasing transistor counts and clock frequencies. An often-quoted law on technology progress known

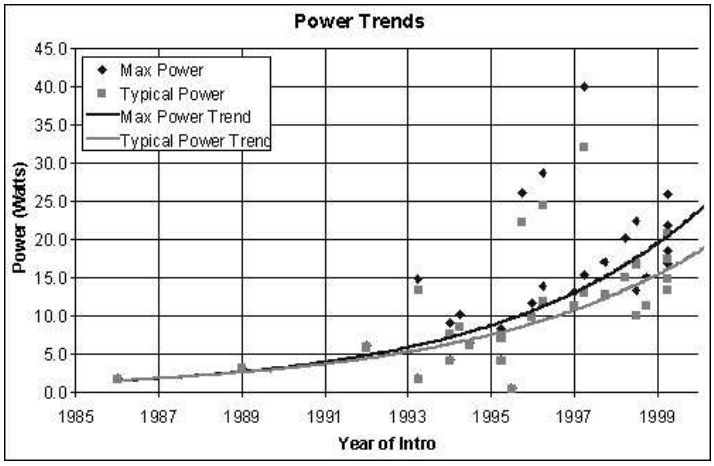


Figure 1.3. The power consumption trends for Intel microprocessors, according to [69]

as *Moore's law* states that the number of transistors on a microprocessor doubles roughly every two years. Power consumption of microprocessors for the PC market has increased significantly. A significant portion of the microprocessor power consumption is not due to arithmetic operation, but consumed by on-chip caches. Current and future microprocessors will spend between 50 to 90 percent of their transistors on cache [103].

The increase in power consumption of typical Intel microprocessors is depicted in Figure 1.3. The estimate given within the International Technology Roadmap for semiconductors [3] is that by 2005 high-performance microprocessors will consume 170 W. When the 2005 data point is combined with the data in Figure 1.3, we observe that an exponential growth in microprocessor power consumption can be expected. Microprocessors for portable devices cannot follow this power consumption trend. The existing power gap between high-performance microprocessors and microprocessors for portable devices is likely to increase significantly. The power gap implies a growing difference in performance and cost between the two types of microprocessors. Users cannot expect equal performance and cost for portable and desktop devices.

1.2.3 Storage

Portable devices require the storage of data, for example, to hold programs, user files, audio tracks, and video clips. This data can be stored in many forms, for example, in fixed read-only form such as a Read Only Memory (ROM), in read/write form such as on a hard disk, or in a small high-speed form such as an on-chip microprocessor cache.

Data storage often consists of a hierarchy, called memory hierarchy, of elements that have different size and speed trade-offs. The fastest storage is located inside the processor in the form of registers. The number of registers is limited and each register stores only a 32 or 64 bit value that can be accessed in a single clock cycle. The second level is the on-chip cache, holding a few KB of memory and taking just one or two cycles to access. The cache itself may have multiple layers; an additional amount of slower cache may be present off-chip. The next level is the main memory, consisting of several MB of storage with an access latency of around 50 ns. Another level may be implemented, consisting of magnetic storage on a hard disk with a multi GB capacity and a latency of 10 ms or more. Instead of a hard disk, flash memory can be used.

Flash memory is considerably more expensive per MB of storage than a hard disk, but consumes

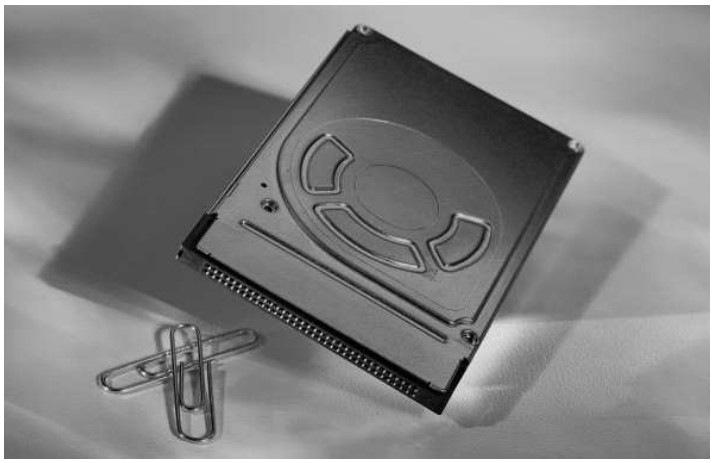


Figure 1.4. The very small hard disk made by Toshiba with a capacity of 5 GB.

significantly less power. Flash memory is functionally and structurally similar to EPROM, but is erasable. Erasing Flash memory requires a time-consuming process of re-writing an entire block. Writing data to Flash has significant overhead because a block erase operation is required. Flash memory is used frequently in portable MP3 music players and digital cameras.

The power consumption of high-performance hard disks is far too high for portable devices. For example, the IBM Utrastar 36Z15 consumes 35 W when starting up to revolve at 15.000 RPM, idle-mode power consumption is 13.5 W. This large power consumption means that portable devices involve considerable compromises. Figure 1.4 shows a portable hard disk made by Toshiba with a storage capacity of 5 GB and weight of 55 gr, measuring only 55 x 86 mm. Compared with the IBM Ultrastar, the Toshiba hard disk has significantly less performance (data transfer rate 20 versus 320 MBps), but it consumes only 1.2 W when spinning up and 0.7 W during idle mode, according to the specifications.

In the last decades hard disk manufacturers have continuously reduced the size of the disks, increased the density, and increased the rotational speed. The smaller hard disk size has a positive influence on power consumption as the energy required to spin a disk is proportional to the square of its diameter. However, the ever-increasing information densities require more sophisticated hardware, increasing both storage capacity and power consumption. The faster rotational speeds give a hard disk more performance in terms of transfer rate, but again also increase power consumption.

1.2.4 Communication

An important aspect of portable devices is the ability to communicate with other devices. Not all devices require this capability, but for others, such as cell phones, it is vital. For portable devices, a simple cable is always the most power-efficient solution. To be truly mobile, portable devices have to use benign air waves to communicate instead of more power efficient conductors such as optical fibers and copper cables.

A multitude of standards exist for connecting devices with both cables and wireless communication. For connecting devices with cables, the RS232, Ethernet, USB, and FireWire standards are popular. Using cables results in a high power efficiency. On the other hand, the existing wireless standards do *not* provide a very power-efficient solution. Improvements are clearly needed to provide both power efficiency and mobility. In the following, we focus on the wireless standards for com-

munication. The multitude of wireless standards can be roughly divided into three classes, each class supporting a different maximum distance between sender and receiver.

The first class of wireless standards has a very short range, in the order of a few meters. Several million of devices sold each year have a short-range infrared link. Phones, PDAs, printers, laptops, etc. use the infrared IrDA standard [91]. The hardware components needed for IrDA with a bandwidth of 4 Mbps cost less than $5. The range of infrared is limited to a few meters and a direct line of sight is required. Another wireless standard, called Bluetooth [174], has been developed to replace all the cables for portable devices in a cost-effective manner. The headset of a phone, or the speakers of an MP3 player can be connected using Bluetooth. The raw bit rate of Bluetooth is just 1 Mbps, insufficient for transferring, for example, video of VHS quality between devices in real-time.

The second class of wireless standards is in the medium range, with a maximum outdoor reach of 500 m. Within this class the IEEE 802.11 Wireless LAN (WLAN) standard [99] dominates the market, with 70 million units sold in 2000. Most of the WLAN units operate within the 2.4 GHz band. This band has become a world-wide license-free frequency, with growing popularity and increasing noise levels. This unregulated band creates a classical economical problem, called tragedy of the commons, where users can gain profit at the expense of others [76]. A number of users use non-compliant wireless equipment to increase their available bandwidth, range, or link quality. The 2.4 GHz band is becoming the equivalent of *the wild west* of the wireless world. In 2001 the first WLAN products with a raw bit rate of 54 Mbps that use the 5 GHz band appeared on the market. As of 2003, this band is still virtually unused.

The third class of wireless standards is in the long range, with ranges in the order of kilometers and nation-wide coverage. Examples of such systems are GSM for Europe and satellite phones. Currently the dominant services on these networks are voice, paging, small text messages, and e-mail. The expectation is that after 2003 the usage of data services will increase. The much anticipated 3rd generation mobile system would bring 128 Kbps wireless streaming video (with low quality) to every phone. Large problems remain because of the lack of applications and the high cost of the 3rd generation wireless infrastructure.

Wireless communication is getting a strong foothold as users want less cables, more mobility, and take the power consumption increase for granted. The technology for wireless communication is developing at a fast pace, especially in the WLAN area. Power consumption is still an issue as technology improvements are often used to increase performance or reduce costs instead of lowering power consumption. The trend in using higher frequencies also tends to increase the power consumption for wireless communication.

1.2.5 User interface

The interaction between the user and the portable device is often expensive in terms of power consumption. We use the term ”user interface” to indicate the components that make this interaction possible. Power management research often concentrates only on the display screen hardware and ignores alternative ways of interacting with the user, such as speech recognition and speech synthesis.

A laptop screen with high resolution and good brightness will drain the batteries quickly. For a typical laptop the screen and backlight are responsible for over 25 % of the power consumption, according to a study conducted by Lorch some time ago [119]. The power consumption of a small display on a PDA is reasonably low, but the readability is also low. Turning the backlight on increases both readability and power consumption considerably. For a Palm Pilot the LCD consumes just 20 mW and the backlight consumes slightly over 90 mW, thus turning the backlight on significantly reduces battery life [40; 55]. New technology such as the head-worn display shown in

Figure 1.5. A low-power near-eye display from Minolta [98].

Figure 1.5 will lower the power consumed by a display significantly and offer excellent readability. The display unit on the eye glasses has a weight of only 5.5 gr, excluding cables. Such head-worn displays have significant advantages such as less power consumption, less weight, and a smaller size. Like headphones, the required cables for connecting the head-worn display are irritating but probably acceptable for end users.

Instead of using a screen it is possible to have the device speak to the user. Dedicated hardware is now on the market for the English language that converts characters into the proper waveforms for driving a headphone speaker. These text-to-speech chips consume only 100 mW and cost less than $10 [189]. The drawback of this approach is that a screen can relay much more information per time unit than speech and only a screen allows non-sequential access.

A keyboard allows for fast user input, but it is a bulky solution. Real-time handwriting recognition requires a powerful processor with considerable power consumption. The Palm Pilot PDA uses a very power-efficient hand-writing recognition mechanism. The user has to write characters both individually and in a slightly different way to reduce the complexity of the recognition process. This system called grafitti uses only 86 mW [40]. Unfortunately, writing by hand in this way is slower than keyboard input. Real-time voice recognition for a number of keywords such as forward, go to, dial, etc. is still manageable on a small processor [74], but unconstrained text input is still error prone and computationally intensive.

Advances in the user interface field are slow. Good head-worn displays and speech input/output could significantly increase the usability of portable devices in the future.

1.3 Problem analysis

The performance advances and power consumption trends as described in the previous sections lead to the following conclusion: No matter how good the battery is, no matter how extremely power efficient the processor is, no matter how efficient the other components are, for competitive reasons companies need to give portable devices so much performance, features, and such a small (battery/device) size that the battery lifetime is still reduced to the minimally acceptable level. In short: power consumption matters and will remain important.

The trends in power consumption for the various components show that they may result in a power

crisis. In order to solve the power crisis, researchers are working on various ways to reduce the power consumption. There are currently three directions in power management.

- Reduce the power consumption of the hardware components
- Improve the techniques to use the sleep modes of hardware components
- Improve the efficiency of the interaction between components

The first direction concentrates on the individual hardware components. Improvements on power efficiency at, for example, the transistor level use techniques such as clock gating, parallel hardware, state machine modifications, and modified memory organization [34]. Reduction of feature size in semiconductor technology provides the basis for many of the advances. This research direction receives significant attention. Researchers in universities and companies are constantly improving the performance and power efficiency of components. As can be seen in the previous section, the improvements for the different components have varying levels of success.

The first research direction is complemented by the second research direction: turning off unused hardware components. For example, if the hard disk has not been used for one minute, it is put into a deactivated state with less power consumption. The rule that decides when to exploit these deactivated states is called a *power management policy*. The deactivated states of hardware components have various names such as sleep mode, doze mode, hibernate mode, suspend mode, or simply inactive mode. Hardware components can support more than one single deactivated state. Each state has a different power consumption and wake-up time. The wake-up time is the time it takes a hardware component to return to the activated state. It ranges from 1 μs to several seconds.

Exploiting deactivated states has been the topic of many research publications. Douglis and Li applied the technique to hard disks and published their classical papers in 1994 [52; 114]; many other publications followed such as [51; 80; 86; 106; 124]. The Advanced Power Management (APM) standard defines the BIOS interface for power management [45]. The OS can send a request to the BIOS to deactivate a certain component. Exploiting deactivated states is now a common feature for laptops with a general-purpose OS. The effectiveness of this direction varies considerably. With long idle periods and reasonable wake-up times, a significant amount of power can be saved. In other cases power is wasted due to the cost of the hardware reactivation, combined with the performance decrease caused by the wake-up time.

A common property of the second research direction is that each power management policy controls the deactivated modes of a *single* hardware component. Also, policies for different devices do not interact and applications are not involved in power management, meaning that there are opportunities for improvement.

The third research direction to tackle the power crisis is improving the efficiency of how the different components work together. In this research direction, applications are included in the power management of components. The way in which components are combined and interact with each other is an important factor in the total power consumption. Improved cooperation between components (including the applications) can yield significant savings according to [138]. The research activity in this area has been started just a few years ago and is still limited to a few publications [55; 122; 126; 139]. To quote Benini: “The development of a communication infrastructure to support application-aware power management is still an unexplored research area” [17]. Only ad-hoc proposals exist to make application knowledge available to the lower layers of a device, for example, the ad-hoc application processor reservation interface in [72]. Likewise, applications do not have any knowledge of the possible trade-offs in the lower layers of a portable system, due to the absence of a generic

information exchange infrastructure. This recently started research third direction has not even been given a name. We propose the name **cooperative power management**.

Within this research direction, components interact by exchanging information about their current workload and environment. Furthermore, components should adapt to changes in both workload and environment to maintain power efficiency. The creation of such adaptive systems has proven to be difficult and much work is still needed; "the design of adaptive mobile systems is currently a black art" [139].

The conclusion of our analysis is that the power efficiency of individual components has improved over time due to the research advances in hardware power efficiency and the exploitation of deactivated modes. However, from a power-management perspective, the structure of a portable device has essentially remained the same. *The model of interaction between the components of a portable device has not been improved in the last decade*. Opportunities exist to further reduce the power consumption by creating a unified interaction infrastructure that facilitates cooperation.

1.4 Approach

Our approach to solve the power crisis in portable devices consists of a unified interaction infrastructure called *Perfd* that realizes cooperation between the individual power management policies.

1.4.1 Cooperating policies

One can increase power efficiency by creating *cooperating policies*. Cooperation between power management policies means that policies are no longer separate entities, but that policies coordinate actions with each other and share information. Multiple cooperating policies work together to efficiently manage a portable device. Each cooperating policy controls *a single* component. Figure 1.6 shows a cooperating policy. The figure shows the two foundations of a cooperating policy: context awareness and sharing of non-functional information. The concept of *context awareness* means having the sophistication to

- to understand one's environment
- interact with that environment
- reacting to environmental changes

Applying context awareness to power management policies means that such policies no longer operate in isolation. Cooperating policies both understand and cooperate with the applications, OS, device drivers, and hardware. Currently, policies are isolated and exclusively located inside the individual components. With context awareness, policies begin to operate on the portable-device level.

A cooperating policy shares *non-functional information*, such as power consumption, future resource requirements, performance metrics, configuration alternatives, current sharing schedule, end user benefit estimation, and environment changes. Traditionally, such information is kept private or is unknown. The applications are key providers of non-functional information, however, they normally do not share it with policies. Applications need modifications in order to share more information; this is a serious drawback. Fortunately, such modifications do not require much effort [56] and most applications for mobile phones or PDAs are already exclusively developed for those portable devices.

Cooperating policies control two activities: scheduling and configuration within a single component, as shown in Figure 1.6. Both will be discussed below.

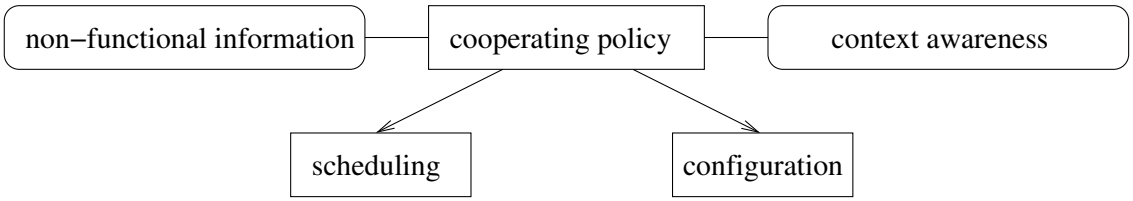


Figure 1.6. The structure of a single cooperating policy within the Perfd framework.

Power-efficient *scheduling* is an important task of a cooperating policy. Scheduling means planning an access pattern when one or more components (clients) are requesting service. In the simple case, the clients are given exclusive access for a number of fixed time slots and scheduling merely consists of assigning components to time slots. In the complex case, the scheduler has several options to deliver the same service and the most power-efficient option must be determined.

The scheduling of the processor component can be complicated when the processor supports a range of frequencies and when tasks can be executed at different processor frequencies. Current approaches to processor scheduling give a good indication of how the scheduling can be improved to increase power efficiency. The *clock scheduling* problem consists of determining the task to execute, the time of a processor frequency change, and the required amount of change. Many modern processors contain mechanisms that improve the power efficiency by decreasing the clock frequency. According to our measurements, the difference in energy per instruction can be as large as a factor of 5 [151]. For example, during DVD playback, the processor needs to deliver more performance than for processing key strokes in a word processor. The OS can use processor load statistics to estimate the processing requirements of applications and solve the clock scheduling problem in isolation. However, the OS has no knowledge of DVD playback deadlines and DVD video frame complexities, and hence this will result in missed frame deadlines because the time to decode each frame varies considerably (bursty). By sharing processing requirements and deadlines between the video decoder and the cooperating policy, the clock scheduling problem can be solved more efficiently, using approximately 24 % less power. This can be deduced from [145].

The sharing of non-functional information can also increase the power efficiency of the *configuration* action. With configuration we mean the selection of operational modes of one or more components. A single operational mode (setting) offers a certain value for properties such as performance, power consumption, and resource usage in general. For example, the processor in the previous scheduling example offers several operational modes; every supported frequency is a different mode.

Another example of configuration involving multiple components is the selection of the operational mode for an IEEE 802.11b wireless LAN component. The IEEE standard defines two operational modes, the normal mode that continuously uses typically 1 W and delivers 5 Mbps [58] and a reduced-power mode with both less power and less bandwidth. When transferring real-time video across such a wireless LAN, the reduced-power mode limits the video bit rate whereas the normal mode limits the battery lifetime. Existing power management standards cannot exploit the possibility to set the video bit rate at the maximum bandwidth supported by the wireless LAN.

Both configuration and scheduling adapt the portable device to change in workload and environment. The difference is the time at which this takes place. Configuration is done prior to service delivery, whereas scheduling is done during service delivery.

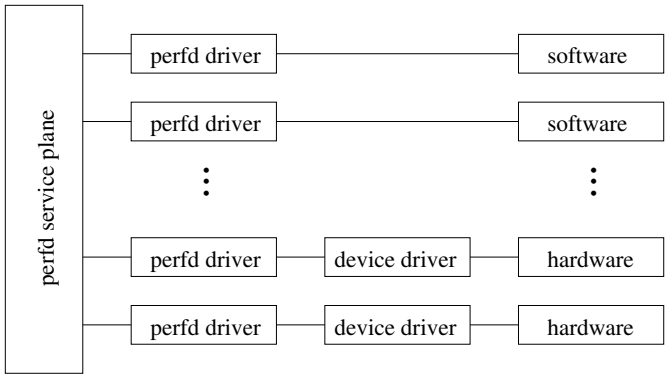


Figure 1.7. The structure of the Perfd framework for global cooperation.

1.4.2 Perfd framework

Creating cooperating policies that work with existing hardware, device drivers, OSes, and applications is not trivial. We have designed a framework for the realization of cooperating policies. It uses a single power management API. This framework is called: *Perfd*, short for Performance Daemon. Perfd enables context awareness and information sharing with a message-passing infrastructure. For all components in a portable device operating within the framework, such as the processor, hard disk, and video decoder, it must be known what their trade-offs are in the various supported operational modes. For example, a video sequence that is decoded at a low frame rate uses less power. The information on such performance versus cost trade-offs is made available to all interested components using a message-passing infrastructure. The trade-offs can be accessed through a single power management API in the Perfd framework. This is a significant advancement because it has proven difficult to design a power management API that is generic enough to unify all possible components (including applications) in a portable device, and yet powerful enough to improve power efficiency. Bellosa described this difficulty in 2000 and recognized the "great demand for a power management API to empower applications to affect their power consumption" [16].

The structure of Perfd is shown in Figure 1.7. The hardware and software components are for example the hard disk, processor, video decoder, speech recognizer, and web browser. Applications may need to be modified to fit within the Perfd framework. Low-level mechanisms for driving the hardware are implemented in a traditional OS device driver. However, the OS device driver no longer determines the access schedule to the hardware; this responsibility is shifted to the Perfd driver. The Perfd service plane handles the sharing of information between components and stores state information from the components. For example, an application that needs to know the power consumption of the wireless LAN sends a request to the Perfd service plane. The Perfd service plane relays this request to the Perfd driver of the wireless LAN and sends the answer back to the application. Each Perfd driver contains a cooperating policy. A Perfd driver also contains information on the trade-offs of the supported operational modes. In the case of the wireless LAN this is the bandwidth versus the power consumption for every supported mode. The trade-offs are based on an explicit *performance model* of the controlled component in the Perfd driver. Using this performance model, the Perfd driver determines the most energy-efficient schedule for the component. This is the policy functionality. The OS device drivers contain the mechanisms to execute this policy. Currently, the knowledge on trade-offs is either not present or not explicit and interwoven with other functionality. Configuration is improved

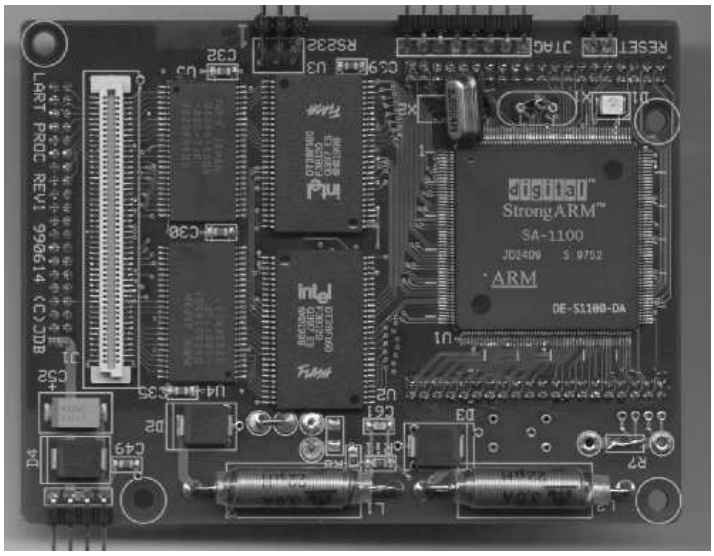


Figure 1.8. The experimental LART platform for power management research.

because the resource usage of several alternative configurations can be calculated using the performance models inside Perfd drivers. The traditional method for configuration relies on guesswork for the configuration, if energy efficiency is considered at all.

Another form of interaction that is supported by the Perfd framework is the reservation of resources. A component sends a request to the Perfd service plane for wireless bandwidth. This request is then delivered to the relevant Perfd driver. The Perfd driver re-calculates the most energy-efficient schedule for the hardware. The energy efficiency of the calculated schedule is determined by the amount of available information. If, for example, the Perfd driver knows that a task can be postponed for a few seconds, the hardware can remain in deep sleep mode. This opportunity for power saving is not available without a cooperating policy.

The Perfd framework not only improves the power efficiency, but also enables new functionality. We have named this new functionality: *guaranteed battery lifetime*. Without cooperating policies the battery lifetime of a portable device is simply estimated by a mechanism that looks at the current battery drain to predict the remaining lifetime. When the Perfd framework is used, however, the power consumption of, for example, the video decoder is known in advance for each operational mode. This knowledge can be used to improve the estimation of the remaining lifetime. The Perfd framework can even guarantee a specified lifetime by selecting a proper operational mode and reacting swiftly to changes in the environment. The Perfd framework requires Perfd drivers for all parts of a portable device. The a priori knowledge from the performance models inside the Perfd drivers will enables us to answer questions like: *what operational mode provides me with the best quality possible and keeps my portable device alive until the end of the movie?*

Experimental validation of the Perfd framework has been carried out as part of this dissertation work. We have implemented the Perfd framework in software and ran it on an embedded platform (called LART, Linux Advanced Radio Terminal) that was developed at Delft University of Technology [9]. This platform is depicted in Figure 1.8. The platform uses a general-purpose embedded microprocessor and runs the Linux OS. All components of a wireless multimedia device shown in Figure 1.2 have been created for this platform.

1.5 Research contributions

Many researchers have recognized the potential of cooperative power management to elevate power management to the next level of technology and also observed the lack of solutions to realize it [16; 17; 55; 139] (all using different terms to denote cooperative power management). We addressed this issue and developed a solution.

The main contributions of this thesis are:

- A unified classification of power management solutions. This classification contributes to the insight in the fundamental principles behind power management. The classification covers both existing power management standards and research proposals (Chapter 2).

- The design of the *Perfd* framework. The framework allows the implementation of cooperating power management policies. This new generation of power management policies is based on information sharing and context awareness. Perfd addresses the need for an infrastructure for communicating about application requirements, resource usage, and non-functional information in general. The innovative concepts of Perfd are a generic power management API, the usage of explicit performance models, and the adaptation architecture for applications (Chapter 3).

- Experimental validation of the Perfd framework. Experiments show power consumption reductions and the guaranteed battery lifetime functionality on actual hardware with realistic multimedia applications. We have enhanced the software of some existing audio and video decoders and created Perfd drivers for them. The decoders now have the ability to interact within the Perfd framework. The decoders can accurately predict their future resource usage, reserve resource in advance, react more swiftly to environment changes, and inform others of the quality versus resource usage cost with a performance model (Chapters 4 and 5).

- A new processor scheduling algorithm. In the specific area of processor power management we have developed a processor scheduling algorithm called *Energy Priority Scheduling* (EPS). The EPS algorithm calculates an energy-efficient setting for modern variable clock-frequency microprocessors. EPS is integrated with the scheduler of the OS and determines which processor schedule uses the minimal amount of energy, in both time and processor speed, but does not violate application requirements. When applications communicate new needs, EPS re-calculates the processor schedule. The overhead of EPS is an order of magnitude less than that of previously proposed algorithms (Chapter 5).

Chapter 2

Power management concepts

THIS chapter introduces the major concepts in power management. Three elements in power management are distinguished: mechanisms, policies, and architectures. All three elements are described in detail. The last section of this chapter presents a novel taxonomy for power management in portable devices. Our presented taxonomy is powerful enough to describe *all* existing power management solutions, both existing industry standards as well as solutions proposed in the literature.

Roadmap

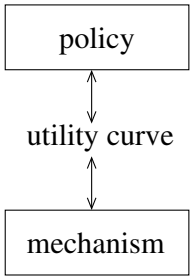


Figure 2.1. The relation between a power management policy and mechanism.

We use a broad definition of power management in this thesis. Power management is manipulating the components of a system to reduce power consumption. As already briefly stated in the introduction chapter, power management is important. For portable devices it will remain important for some time in the future because every improvement in power efficiency is countered by an increase in the number of features or performance, or in a reduction of the (battery) size and weight. We call this the *power consumption balance* principle. With a reformulation of the problem, the great importance of power management becomes clear. The improvements of features, performance, and reduction of the (battery) size and weight *depend* on power efficiency advances.

A prime example of the power consumption balance principle is the mass market of x86 laptops. Power efficiency improved with standards for power management such as APM [45] and ACPI [90] in combination with more efficient policies. However, the power consumption of laptops has not decreased over the years. When more efficient hard-disk policies are discovered, they are *not* used to reduce the average power consumption, but employed in the next hard-disk model to increase storage capacity and to reduce access latency with compatible power usage. Every improvement in the power efficiency of hardware components themselves is also neutralized. In 1993, a typical display had no color, a resolution of 640 by 480 pixels, and dissipated 2 W, according to [78]. This publication by senior industry specialists predicted an aggressive improvement in battery lifetime. In two years, the battery lifetime of a typical laptop would increase from roughly 4 hours towards typically 8 hours due to technology advances in displays and other components. A decade later this prediction has proven to be false because consumers prefer a light laptop with a high-resolution color screen instead of a doubled battery lifetime.

Power management solutions often divide the power management problem into more manageable pieces by treating each component in isolation. Furthermore, each component is managed by two entities: a power management mechanism and a power management policy. The policy determines the manipulation of a component and the mechanism contains the intelligence to do the actual manipulation. The relation between a mechanism and policy is illustrated in Figure 2.1. In the simplest case, a power management policy detects that a hardware component is idle and sends a sleep command to the power management mechanism of that component.

A more advanced approach is to be aware of trade-offs in the components and to exploit these to improve power efficiency. Such trade-offs can be made explicit through a *utility curve*. A utility curve of a component specifies the relation between performance (or quality), power consumption, and other costs such as wireless spectrum usage. Each point on the curve represents an *operational mode*. The metric of performance or quality can be multi-dimensional. For example, the quality of

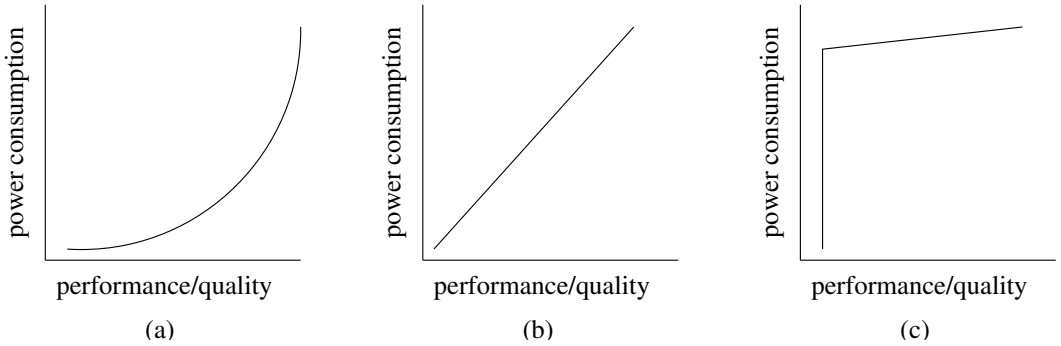


Figure 2.2. The power usage with increasing performance or quality for three common utility curves, exponential (a), linear (b), and flat (c).

an operational mode of a video decoder may be quantified by a combination of temporal frequency in frames per second, spatial resolution in pixels, and perceptual quality such as "broadcast quality", "VHS quality", or "cam-corder quality". Utility curves are a relatively new concept and the documentation of hardware components usually does not explicitly describe them. If a policy had access to an explicit description of the utility curves, it could make better trade-offs to improve power efficiency.

Figure 2.2 shows three common types of utility curves. Figure 2.2.a shows the exponential type of utility curve where low performance or quality means a significant lower power consumption than an operational mode with high performance or quality. An example of such a utility curve is a processor that supports voltage scaling. The difference in power consumption per processor frequency can be very large. The linear type of utility curve is depicted in Figure 2.2.b; a performance increase results in a power consumption increase of equal proportion. A 802.11 wireless LAN card has such a utility curve; more bandwidth costs proportionally more power. Figure 2.2.c shows the utility curve of the flat type, where the activation of the hardware is costly and the subsequent usage cost is marginal. For example, rotating a hard-disk platter for one minute is relatively expensive in terms of power consumption; whether light or intensive data transfers take place during that time makes little difference.

In principle trade-offs such as processor performance versus power consumption are continuous, hence the name utility *curve*. However, a number of factors (e.g. product cost and design complexity) limit the number of operational modes in commercial hardware implementations. For example, Intel has limited the number of supported processor frequencies to just two with their SpeedStep technology [89]. SpeedStep provides simply an economy mode and a performance mode for the processor, reducing the additional complexity and cost of the processor to a minimum.

There also exists an economic phenomenon that influences utility curves. We call this the *mechanism/policy interlock*. In the highly segmented PC industry, hardware components are produced by a different company than software components. The hardware company is not strongly motivated to introduce new power-saving mechanisms when no policy exists to exploit them. On the other hand, a software company is only motivated to develop a policy for (experimental) mechanisms when a sufficient number of consumers have bought that hardware. This classical chicken-and-egg problem does not exist in single-purpose devices such as a cell phone where a single vendor develops both mechanism and policy. Parallels exist with the problems of hardware / software co-design [127], where the hardware (mechanism) cannot be independently developed from the software (policy).

The mechanism/policy interlock illustrates the close relation between mechanisms and policies. Policies and architectures are the main topic of this thesis, but without more mechanisms their effi-

ciency gains will be limited. Future portable devices need both smarter policies and more mechanisms to increase power efficiency.

2.1 Mechanisms

Within Section 1.2, the trends in power consumption and technology have been analyzed in general terms. This section describes in detail the trade-offs that can be made in hardware to reduce power consumption. A large power saving can be realized by simply deactivating a hardware component. Deactivation is always possible for hardware components and requires no further explanation. Within this section, only other mechanisms are discussed.

2.1.1 Batteries

Batteries are the dominant technology for powering portable devices. Unfortunately, the possible influence of a run-time power management policy on battery performance is somewhat limited.

Batteries have two important properties when in use: their voltage and their storage capacity. An ideal battery has a constant voltage throughout a discharge, which drops instantaneously to zero when the battery is empty, and has an energy storage capacity that is independent of the discharge rate [130].

In practice, batteries are far from ideal and the voltage gradually drops during the discharge due to changes in the active materials and reactant concentrations [117]. An important non-ideal battery property is the *rate capacity effect*. The energy stored in a battery (capacity) depends on the power consumption load. The C rating is used in the battery industry to normalize the load current to the battery capacity [117]. For example, a load of $0.1C$ for a battery with a C rating of 1 A-hours is 100 mA. For a simple NiCd battery, the energy storage capacity decreases by about 40 % over a range of discharge rates between $0.1C$ and $10C$. Thus, high power consumption results in reduced battery energy.

The *recovery effect* is another non-ideal battery property. Recovery occurs when a battery is discharged for short time intervals and each discharge is followed by an idle period. Recovery can significantly improve the capacity of a battery [62].

Some battery types suffer from the *memory effect*. Repeated partial discharges reduce the battery capacity over time for these type of batteries. The memory effect can be countered by performing regular deep discharges.

An accurate performance model of battery capacity is presented in [53] using partial differential equations to represent the fine-grained electro-chemical phenomenon of cell discharge. However, such accurate performance models are computationally expensive and take numerous days of calculations on a portable device such as a PalmPilot.

In [144], simulation experiments are described in which a processor workload is adapted to exploit non-ideal battery properties (a policy). A stochastic model is presented that estimates the capacity of a battery under a specific discharge profile. The model takes both the rate capacity and recovery effect into account. A battery lifetime extension of a factor of 4 is reported for their (synthetic) workload. However, a single capacity calculation still takes several seconds. This makes it unsuitable for run-time use, because run-time evaluation of the capacity of a dozen possible discharge profiles would be prohibitively expensive.

To conclude, the discharge profile determines the battery capacity, but it is difficult to accurately calculate battery capacities using simple and time-efficient algorithms.

2.1.2 General-purpose processors

The power consumption of general-purpose processors is increasing. Figure 1.3 showed the exponential increase in power consumption for Intel processors from 1985. The initial response to the low-power demand was to lower the supply voltage of a processor, for example, by reducing the supply voltage from standard 5.0 V to 3.3 V, power was reduced by 56 %. Voltage scaling is a more general technique to vary the processor voltage at run time. However, lowering the voltage increases the circuit delay, hence at lower voltages the clock frequency of the processor must be reduced. By using voltage scaling, a processor obtains an exponential utility curve.

Voltage scaling principles

We will now introduce the basic principles of power consumption in processors and the effects of voltage scaling. For digital CMOS circuits, used in the majority of microprocessors, the power consumption can be modeled quite accurately by simple equations [26; 92]. CMOS circuits have both dynamic and static power consumption. Static power consumption is caused by bias and leakage currents. It is neglected in most designs that consume more than 1 mW.

The dynamic component is the dominant source of power consumption for CMOS microprocessors. Every transition of a digital circuit consumes power, because every charge and subsequent discharge of the digital circuit's capacitance results in dissipation in the circuit's resistive components. According to [26], the dynamic power consumption can be estimated by

$$P_{dynamic} = \sum_{k=1}^{M} C_k \cdot f_k \cdot V_{DD}^2 \tag{2.1}$$

where M is the number of gates in the circuit, C_k the load capacitance of gate g_k, f_k the specific switching frequency of g_k, and V_{DD} the supply voltage. It follows from Equation (2.1) that reduction of V_{DD} is the most effective way to lower the dynamic power consumption. Lowering V_{DD}, however, creates the problem of increased circuit delay. An estimation of circuit delay is given by

$$\tau \propto \frac{V_{DD}}{(V_G - V_T)^2} \tag{2.2}$$

where τ is the propagation delay of the CMOS transistor, V_T the threshold voltage, and V_G the input gate voltage [92]. The propagation delay restricts the clock frequency in a microprocessor. From Equations (2.1) and (2.2) it follows that there is a fundamental trade-off between switching speed and supply voltage. Processors can operate at a lower supply voltage, but only if the clock frequency is reduced correspondingly to tolerate the increased propagation delay. When we assume that the dynamic power is dominant and the gates g_k of the microprocessor form a collective switching capacitance C with a common switching frequency f, we obtain

$$P = C \cdot f \cdot V_{DD}^2 \tag{2.3}$$

Equation (2.3) shows that a clock frequency reduction linearly decreases power, and that voltage reduction results in a quadratic power reduction. The critical path of a processor is the longest path a signal must travel in a clock cycle. The implicit constraint is that the propagation delay τ of the critical path must be smaller than $\frac{1}{f}$. In fact, the processor ceases to function when V_{DD} is lowered and the propagation delay becomes too large to satisfy internal timings at frequency f (Equation 2.2). Voltage scaling is the mechanism to minimize power consumption for a given clock frequency.

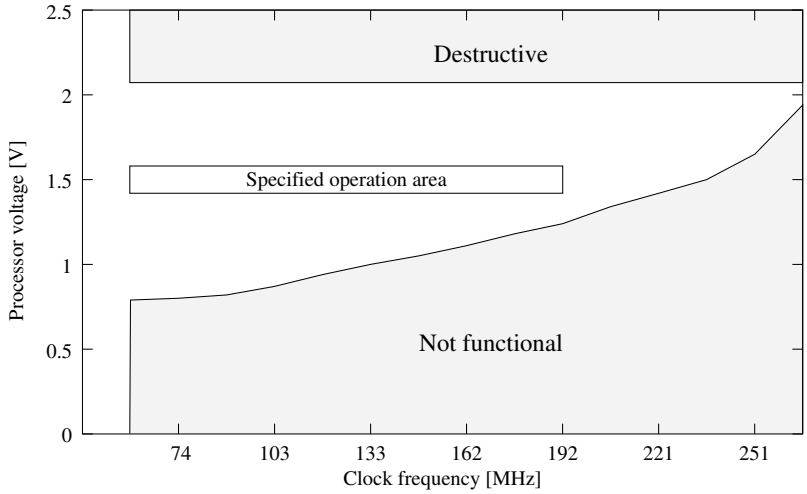


Figure 2.3. Processor envelope.

Implementations

To put the previous formulas into perspective, we will discuss a number of experimental and commercial systems. In 1996, one of the first papers was published that describes an actual hardware implementation using voltage scaling [33]. This implementation applies voltage scaling to MPEG video decoding on a DSP. The frequency and voltage are adjusted to match the varying complexity of video frames. In [94] a dedicated cryptography processor is presented that uses voltage scaling. When running at 50 MHz, this processor requires a supply voltage of 2 V and consumes at most 75 mW; at 3 MHz a supply voltage of only 0.7 V is required and the power consumption drops to a mere 525 μW.

In 1998 the first experimental results on a general-purpose processor were published [107]. The architecture of a R3900 RISC core was enhanced with a critical path replica to measure the minimally required supply voltage. The RISC core operates on 1.9 V at 40 MHz and on 1.3 V at 10 MHz. All intermediate frequencies are also supported. This first general-purpose implementation did not have a full chip set and lacked an operating system. In 2000, Grunwald et al. presented experimental results on a complete general-purpose platform called Itsy, running the Linux operating system [68]. Itsy uses a standard commercial StrongARM SA1100 processor that supports voltage scaling. The savings by the Itsy are very modest because only two voltage levels have been implemented, 1.5 V ($\geq$ 162 MHz) and 1.23 V ($<$ 162 MHz). The resulting difference in processor power consumption between the two levels is only 15 %. Better results are obtained with the SmartBadge platform [171], which is similar to the Itsy. Extensive power measurements on real-time MP3 audio decoding and MPEG video decoding show that an energy reduction of 40 % is possible on this platform. An enhanced version of the Itsy supporting speech recognition, X windows, audio decoding, MPEG video, and again two voltage levels is presented in [74]. Burd and Pering designed and implemented a processor capable of voltage scaling based on an ARM8 core [27]. Their processor is fabricated in 600 nm technology and uses aggressive power-saving features. In high-performance mode it runs at a speed of 80 MHz and consumes 476 mW at 3.8 V. When running at 5 MHz and 1.2 V, the processor only consumes 3.24 mW. Thus, power consumption is reduced with a factor 147, while performance drops with a factor 16. In other words, the energy per instruction is reduced with a factor of 9.

In parallel with the above projects we have created our own portable platform for voltage scaling

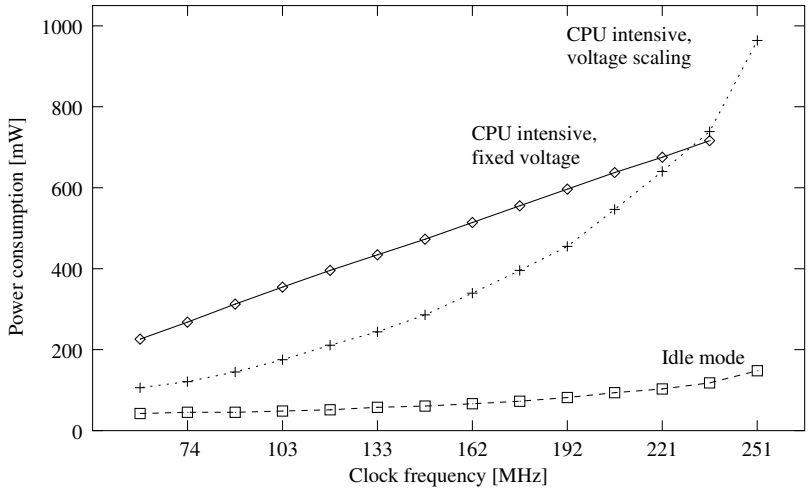


Figure 2.4. Total power consumption for idle and cpu-intensive workloads.

research [9]. Our platform is somewhat similar to Itsy; we also use a standard SA1100 processor and run Linux. Our platform is called Linux Advanced Radio Terminal (LART) (Figure 1.8), and described in Section 1.4.2. Figure 2.3 shows the processor envelope for the different frequencies that are supported by the StrongARM processor. The LART uses a 7-bit DA converter and supports 128 different supply voltage levels. A supply voltage of 0.79 V is sufficient for the processor running at 59 MHz. A frequency of 251 MHz requires 1.65 V. These supply voltages are outside the manufacturer's specifications of 1.5 V. All processors we used were able to run at these voltage and frequency combinations. A number of destructive tests indicated that the maximum frequency of the SA1100 processors is roughly 265 MHz, significantly beyond the official specified maximum of 190 MHz.

We measured the effect of voltage scaling on the power consumption of the complete LART platform, including memory, voltage conversion, etc. Figure 2.4 shows the total power consumption of the LART under two different workloads: Idle and CPU-intensive.

The Idle workload measures the background power consumption of the LART, which is always spent regardless of the processor load. The Linux scheduler puts the processor into halt mode when no processes are active. Halt mode stalls the CPU, but other services of the embedded processor such as the memory controller and internal timer are still operational [88]. All these services are driven by the processor clock, which explains why the power consumption in halt mode increases with the frequency. The SA-1100 also supports a more power efficient sleep mode, but this mode interrupts DMA transfers, stops the LCD controller, blocks memory access, etc. Also the wake-up sequence takes much longer than in halt mode, compromising responsiveness.

The CPU-intensive workload consists of the Dhrystone benchmark utilizing both the CPU and the cache. We first measured the effect of scaling the clock frequency while keeping the voltage constant at 1.5 V. In this case the power consumption increases linearly with the frequency, as is expected. Next, we measured the power consumption when the core voltage is set to the minimal value reported in Figure 2.3. The resulting curve shows the expected exponential increase of power consumption when the frequency is varied from 59 to 251 MHz. The CPU-intensive workload curve indicates that voltage scaling produces an exponential utility curve.

From the power consumption at 59 MHz (105.8 mW) and at 251 MHz (963.7 mW) it follows that an instruction at peak performance consumes a factor 2.1 more energy than at lowest performance.

When we neglect the non-CPU subsystems of the LART, which are supplied from a *fixed* 3.3 V, and focus on the CPU, the power consumption is 33.1 mW at 59 MHz and 696.7 mW at 251 MHz (not shown). The raw CPU energy/instruction difference is thus a factor 4.94

Voltage scaling is moving from the research field into the commercial market place of embedded and x86-compatible processors. AMD, Transmeta, and Intel currently provide processors with voltage scaling. The mechanism/policy interlock influences the availability of voltage scaling. Intel introduced voltage scaling for the laptop market with SpeedStep [89] in January 2000. The SpeedStep policy is fixed in the hardware: when the batteries are used, the low speed and voltage is selected. When the laptop user switches from batteries to AC power from a wall socket, SpeedStep switches the speed and voltage. The OS is not aware of the changes in the processor frequency and voltage. AMD added voltage scaling capabilities to the AMD K6 processor family in April 2000. The AMD policy is more advanced but requires enhancements to the laptop OS.

As can be deduced from the increasing support in commercial processors, voltage scaling is becoming a mainstream technology. Research on the policy side however is still needed because this mechanism is still not exploited to its full potential.

2.1.3 Wireless LANs

A wireless link is responsible for the transfer of information with minimal or no corruption. The IEEE 802.11b Wireless LAN (WLAN) standard [99] defines a method to transfer several Mbps across a distance of up to a few hundred meters (See Section 1.2.4). Several products, mostly PC Cards, are available that conform to this standard.

To save power, the 802.11b standard defines a method to periodically activate WLAN cards. By using this method the card can be effectively deactivated more than 90 % of the time. This periodic activation feature requires a base station. A WLAN card informs the base station of its decision to deactivate and states the interval between the periodic wake-ups. A typical interval lies between 100 ms and 10 s. The base station buffers all packets destined for the WLAN card. Periodically the base station broadcasts a packet containing information about buffered packets. The buffered packets are listed in the 802.11b Traffic Indication Map (TIM) [190]. When the WLAN card wakes up, it listens to the TIM and requests packets destined for itself, if any. Periodic activation can result in savings up to 80 % according to actual measurements [105].

Sending bits through the air makes them vulnerable to loss and corruption. Corruption of bits is specified using the Bit Error Rate (BER). When bits within a TCP/IP packet are corrupted, the packet is dropped. The Packet Error Rate (PER) is also an important measure for the performance of a wireless link. With a BER of $1.9 \cdot 10E^{-6}$, TCP/IP throughput is significantly reduced [12]. For comparison, a fiber optic cable typically has a BER of around $1 \cdot 10E^{-12}$.

In 802.11b there are two parameters that can be set from the software at run time: the bit rate (1 to 11 Mbps) and RF transmit power (0 to 20 dBm). However, some implementations fix the RF transmit power to the maximum. These parameters determine the power consumption and performance.

In general, channel bandwidth (in Hz), RF transmit power and noise place an upper bound on the capacity (in bps) of a wireless link [147]. The theoretical limit is given by the important channel capacity theorem by Shannon [166]:

$$C = B \cdot \log_2(1 + S/N) \tag{2.4}$$

where C is the channel capacity in bits per second, B is the channel bandwidth in Hertz, S is the signal strength in Watts, and N is the noise power, also in Watts. For an ideal receiver with a sufficient signal to noise ratio of a signal, the BER will approach zero if the bit rate is well below the channel capacity.

Another key parameter in a wireless link is the distance between receiver and transmitter. When radio signals are transmitted in free space, power density falls off as the square of the range. This effect is due to the spreading of the radio waves as they propagate. A basic model of free space signal loss defines the loss (in dB) as:

$$L = 20 \log_{10}(4\pi D/\lambda) \tag{2.5}$$

where D is the receiver-transmitter distance, and λ is the free space wavelength ($\lambda = c/f$, where c is the speed of light, and f is the signal frequency in Hertz). Equation 2.5 only describes the signal loss due to free space propagation. The signals from a typical WLAN card are also obstructed by, for example, the laptop casing, walls, and ceilings. The signal strength loss due to obstruction can be significantly higher than the loss due to the receiver-transmitter distance. The radio signals may also be reflected by objects and reach the receiver via a longer path. For example, the wireless link of a cellular phone conversation may follow a path that first reflects on a building and then changes to a direct line-of-sight when the user walks across the street. Signals that have traveled along different paths can interfere destructively at the receiver. Signals traveling the shortest path will arrive first. A completely out-of-phase signal can even reduce the received signal to zero. This effect is known as multi-path interference. Multi-path can reduce the signal strength indoors with 30 dB [147]. A significant portion of the power consumed by a wireless link is taken up by hardware to overcome the multi-path problem.

The 802.11b standard uses a technique known as direct sequence spread spectrum to combat the multi-path effect. By spreading out the signal over a larger part of the spectrum, a degree of multi-path resistance is created. The 802.11b standard uses the trade-offs in Equation 2.4 and 2.5 to create a wireless link with a low BER. The maximum bandwidth that can be used at 2.4 GHz is set at 22 MHz per band. The maximum radio signal output power is limited by European regulation to 100 mW. 802.11b products currently always use the maximum bandwidth and maximum output power. 802.11b WLAN cards indirectly measure the noise power. The *automatic rate selection* feature selects the highest possible bit rate (1, 2, 5.5, or 11 Mbps) that the channel can sustain given the current noise power.

Automatic rate selection may produce a power-inefficient trade-off between the radio signal output power and the bit rate. Automatic rate selection simply maximizes the bit rate without *any* consideration for power consumption. The automatic rate selection feature can be turned off. However, this requires manual negotiation of the bit rate between the receiver and transmitter. Automatic rate selection is implemented in the firmware of 802.11b PC card products and cannot be easily replaced with a more power efficient solution. Output power has a large impact on the power consumption of a 802.11b PC card. Creating 100 mW of output power results in at least 200 mW of power consumption for the analog 2.4 GHz components. In [54], simulation results are shown where the 802.11 output power is reduced. The utility curve for 802.11 hardware is complicated because the output power and bit rate both influence the receiver-transmitter distance and the BER.

There are wireless standards somewhat similar to 802.11, such as Bluetooth. The Bluetooth standard [174] is designed for low-power operation. The range of Bluetooth is limited to 10 meters with 1 mW of output power. The standard supports one bit rate (1 mbps) and has no mandatory support for output power modifications. The utility curve is therefore not very useful; only sleep modes can be exploited.

To conclude, mechanisms in wireless LANs exist to trade off power versus performance, however, their usage in the 802.11b products that are on the market in 2003 is still problematic due to the automatic rate selection feature.

Component	Mechanism	Description
batteries	rate capacity	lower power consumption increases energy storage
	recovery	intermittent discharge/idle periods increase energy storage
processor	voltage scaling	lower processor speeds are more energy efficient
WLAN	less RF output power	less power reduces range and increases BER
	bit-rate reduction	lower throughput for more energy efficiency
LCD display	backlight dimming	reduced readability to save power

Table 2.1. Summary of different trade-offs in power management mechanisms.

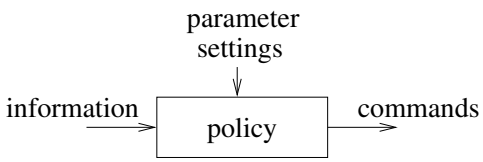


Figure 2.5. A general model of a power management policy.

2.1.4 User interface

It is unlikely that with the current LCD displays there is much opportunity for a policy to play a part in lowering power consumption with fine granularity. For most LCDs the biggest power consumer is the fluorescent backlight that affects the whole screen [111]. At the coarse grain the backlight intensity may be varied depending on environmental conditions and user activity to save power. With reflective screens, field emission screens, and plasma screens, the energy consumption is a function of the intensity of the screen pixels. Software may be able to selectively set the intensity of groups of pixels, depending on the user activity. However, support for this selective pixel illumination feature cannot yet be found in mainstream displays.

2.1.5 Summary

Table 2.1 gives a summary of the different power management mechanisms. Each of the previously discussed mechanisms is listed with a short description. Internal memory is not listed in this table because there are no trade-offs beyond simple deactivation. It is possible for an OS to dynamically deactivate memory, depending on the current applications demand. However, there are currently no known implementations of general-purpose OSs that support this feature and it will take significant effort to implement it.

2.2 Policies

This section introduces power management policies. An understanding of policies is required for the power management taxonomy presented at the end of this chapter.

We define a power management policy as an algorithm that controls one or more power management mechanisms. A simple policy can control a basic mechanism, such as the hard-disk on/off. To exploit trade-offs between two mechanisms, such as the processor versus the wireless link, advanced and cooperative policies are needed.

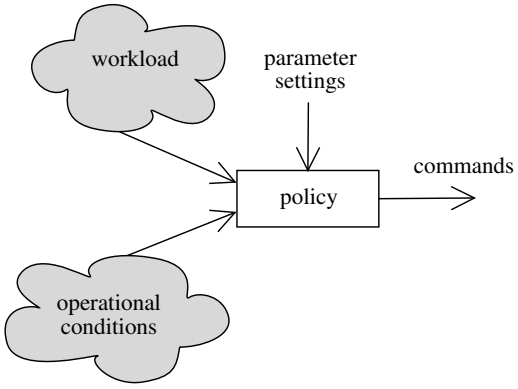


Figure 2.6. The concept of workload and operational conditions as policy input information.

Figure 2.5 shows a general model of a policy. A power management policy itself consists of algorithms that use input information and parameter settings to generate commands to steer the mechanisms. The possible range of input information for a policy can be very diverse, ranging from hard-disk usage statistics to user movement patterns. The commands drive the mechanism(s) under control of a policy. A command can be a simple on/off directive or a more complex directive. Parameters settings influence the behavior of a policy, for instance, a hard-disk sleep time-out setting.

An example of a policy that fits this model is a hard-disk policy. A hard-disk policy algorithm receives as input the history of disk requests and estimates the likelihood of an upcoming disk request. When no disk requests have been issued recently, it is assumed that the hard-disk is not required for a while and the read head is deactivated to reduce power consumption. When no disk request are issued for a longer period, more aggressive sleep modes are used, for instance, deactivation of the hard-disk spindle motor, buffer memory, and interface hardware.

2.2.1 Functionality

The objective of a power management policy is to maximize performance and to minimize power consumption. This implies that a policy must have knowledge about performance and power consumption, or resource usage in general.

Figure 2.6 shows two types of input information in a more detailed policy model. We define operational conditions as knowledge about the utility curve of the mechanism of a component. Workload is defined as the amount of work expected to be carried out by a component in a certain time period. The workload and operational conditions are often *not* explicitly communicated to a policy and must be derived from other sources of information. Both the workload and the operational conditions can vary significantly in time. A policy combines the information on the application needs (workload) and the operational conditions to derive an optimal match. The functionality of a policy is thus threefold. First, estimate the workload. Second, estimate the operational conditions. Third, find the setting for the controlling mechanism that services the predicted workload while taking into account the operational conditions and power consumption. We will illustrate by means of some examples the influence of input information on the trade-offs to be made in power consumption policies.

The power efficiency of a policy depends partly on the quality of the input information. Often a policy has access to only a limited amount of workload information from which the higher layer

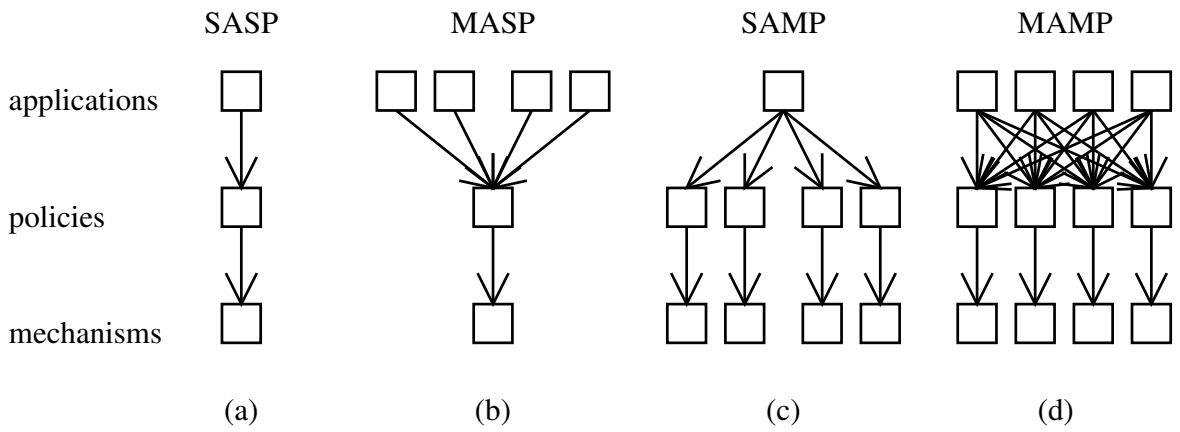


Figure 2.7. Four possible structures of a power management solution.

(applications) demands must be extracted. For example, a policy that controls the processor speed needs to balance the performance of the application with the power consumed by the processor; yet applications do not specify their demands explicitly. When the workload prediction is off by *just a few percent* the processor is running at a frequency that is either too low or too high, resulting in reduced performance or in higher power consumption, respectively.

For some mechanisms the utility curve changes rapidly. An efficient policy uses this information to reduce power consumption. For example, a WLAN policy needs to deal with a constantly changing environment. The cost of transmitting a data packet may vary significantly. A policy can defer packet transmission when transmission costs are high.

For mechanisms without rapid changes in the utility curve, the operational conditions are still important for the power efficiency of a policy. For instance, in the case of a DVD-drive policy, the optimal strategy is often unknown. For DVD movie playback, a possible strategy is to buffer the multimedia content in memory and deactivate the drive for extended periods. Unfortunately, it is impossible to determine the optimal strategy without access to the operational conditions that state the cost of activating and deactivating the drive.

A hard-disk policy is a good example of a trade-off between performance and power consumption with only a limited amount of input information. Current applications do not specify their hard-disk needs explicitly. A hard-disk policy can only use previous hard-disk requests to estimate application needs. The objective of the policy is to have the disk spinning at full speed when a request arrives. The operational conditions indicate that sleep modes are orders of magnitude more power efficient and must be invoked when the probability of an imminent hard-disk request is low. Such a policy must constantly balance the performance penalty of a disk spin-up versus the power efficiency gain of a deactivated hard-disk.

2.2.2 Multiplicity

Only a fictional portable device has one mechanism to control, a single policy, and just one active application. In reality a multitude of policies, mechanisms, and applications co-exist.

Figure 2.7 shows four possible structures for power management. Figure 2.7.a shows the most basic case of power management, the Single-Application Single-Policy (SASP) type. Within this structure the power management problem is relatively simple. Unfortunately, for the majority of cases it is not useful. Figure 2.7.b shows the Multiple-Applications Single-Policy (MASP) type. The

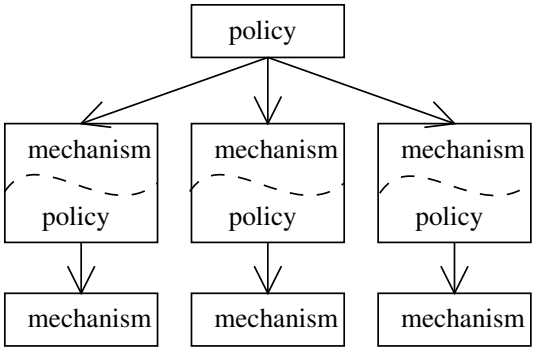


Figure 2.8. A possible layered structure of policies and mechanisms.

complexity is increased because several applications are involved in the power management problem. Figure 2.7.c shows the Single-Application Multiple-Policies (SAMP) type. Dedicated portable devices without a general-purpose OS often have such a structure. For example, a portable GPS receiver and a portable DVD video player may have a SAMP structure. The structure that is used throughout this thesis is shown in Figure 2.7.d, the Multiple-Applications Multiple-Policies (MAMP) type. The MAMP structure is the most suitable one for a portable device, but it is also the most complex type.

Power management solutions proposed in the literature often concentrate on single policy/mechanism combinations and many view a portable device from a MASP or even SASP perspective. It is important to realize that multiple policies co-exist within a portable device and that they can either cooperate or interfere with each other. Thus the opportunity of saving additional power by employing cooperation is often neglected in literature.

A drawback of the MAMP structure is its imposed limitation concerning the ordering of the policies. Every policy is equal and policies are placed side by side, there is no hierarchy. By using a layered structure it is possible to tackle complex problems more efficiently. Policies on the highest level make the top-level decision, while policies at the bottom of the hierarchy decide on the details. Such a layered structure provides a higher degree of scalability.

In a layered structure, policy/mechanism combinations are stacked. Software components such as a video decoder contain both mechanism and policy functionality. Towards the upper layers a video decoder may export a mechanism with a trade-off between power consumption and video quality; towards the lower layers the video decoder may contain a policy for hard-disk usage. Figure 2.8 shows a layered structure where three components are controlled by one top-level policy; each of these three components contains both a mechanism and a policy.

2.2.3 Constraints

Many solutions for power management are described in the literature, each with its own distinct balance of implementation effort and power efficiency gain. The process of creating a power management policy is constrained by several factors, namely, the available *mechanisms*, the policy *input information*, and the *architecture* of the portable device. Figure 2.9 shows these three factors that together significantly limit the number of possible alternatives for a policy.

A mechanism with an exponential utility curve requires a different policy than a mechanism with a flat utility curve. If a mechanism only consists of a single on/off setting, the most simple policy will still function. Only an advanced policy can efficiently control a mechanism that gives access to advanced trade-offs such as delivering the same service with several different resource usage patterns.

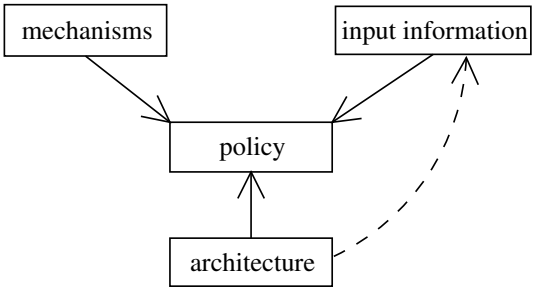


Figure 2.9. The three factors that constrain a power management policy.

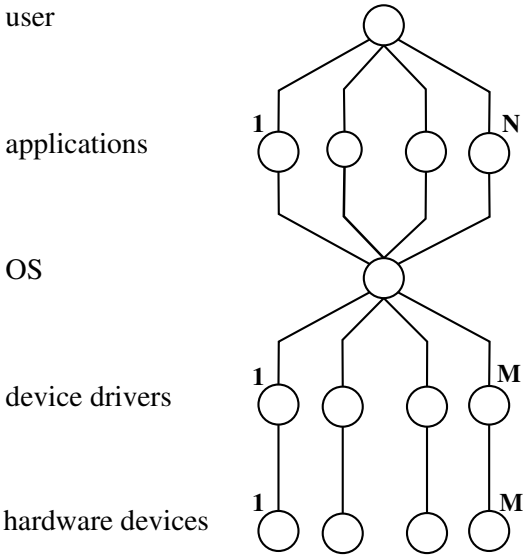


Figure 2.10. Abstract architectural view of the five layers in a portable device.

The available input information constrains the effectiveness of a policy, as already briefly discussed in Section 2.2.1. A policy that is deprived of all input information can only blindly control a mechanism and is very inefficient. A policy with access to very little information has a low power efficiency. For instance, a hard-disk policy that only knows the start and end times of previous requests can only use extrapolation to exploit sleep modes. It can improve the accuracy of its extrapolation when given more information on the applications that issue the requests, such as their identity and current status (running, blocked, terminated). Even more energy can be saved when applications using the hard-disk indicate their future requests prior to using the hard-disk.

The *amount* of input information is not the only influential property of input information. For classical problems such as sorting the items in a list or finding the shortest route in a graph, all input information is available, complete, and accurate. The problems in power management policies are fundamentally different in this respect. A power management policy *always* deals with information that is missing, incomplete, and inaccurate. The power efficiency of a power management policy is dependent on both the sophistication of the internal algorithms and the accuracy, amount, and timeliness of the information available to the policy. We introduce a new term to denote the degree of accuracy, amount, and timeliness of policy input information, namely *input richness*.

The architecture of a portable device poses another constraint on a policy. Figure 2.10 expands

the general MAMP architecture by showing the user, OS, device drivers, and hardware devices.

An important example of architectural influence is the x86 PC architecture. The x86 PC architecture places significant constraints on policies through standards such as IDE. A laptop hard-disk that does not conform to the latest IDE standard is not commercially viable. Support for the IDE standard implies that the hard-disk is directly functional; there is no need to install a device driver for an IDE hard-disk. This constrains the power management policy. Either *all* power management must be handled in the hard-disk firmware or the policy must be placed in a generic IDE policy that has no knowledge of the trade-offs and wake-up times for specific hard-disk models.

A power management policy can be located inside any one of the various (sub)layers in the overall architecture. This *policy placement* influences the input richness of the policy, because the architecture can severely limit the information that each (sub)layer "sees". For example, in a wireless audio streaming scenario, each layer has access to different information. The analog hardware in a WLAN has access to the quality fluctuations with nanosecond accuracy. The 802.11 link layer implemented in firmware can only see the quality fluctuations for each packet through wireless re-transmissions. The TCP layer has even less information and can only observe the re-transmission across the whole connection, but does not know if the re-transmission is caused by network congestion or bad wireless conditions. Interfaces to exchange information between the various (sub)layers in the architecture are therefore important for improving the input richness at various places in a portable device.

2.3 Policy properties

This section provides an extensive analysis of important characteristics of policies. The characteristics of policies are abstracted towards several properties. Each property has several possible values, each subsection discusses the different proposed values in literature. The policy properties are approach, input information, mechanism control resolution, time of fixation, and interaction model. These generic properties are used later on as the basis for our power management taxonomy in Section 2.5.

2.3.1 Approach

Numerous approaches to power management policies are proposed in the literature. All approaches use one or a combination of the three elementary approaches listed in this subsection.

Time-out

The most basic implementation of a policy is observing when a hardware component is idle and deactivating it after a fixed time-out period. This basic approach is called a *static time-out*-based approach. In 1994 two early papers were published that measured the effectiveness of such an approach for hard-disks [52; 114].

Deactivation of a hardware component during an idle period is not always power efficient. There is a tension between power efficiency and performance. In the case of a hard-disk, the wake-up time is considerable and frequent deactivation has a noticeable impact on performance. For an elaborate analysis of time-out-based policies, see [21]. The time required for reactivation of the component may be significant compared to the idle period and hence effectively slow down incoming requests. The cost in terms of energy to deactivate and subsequently activate may be larger than the savings obtained during deactivation. The minimum idle period required to compensate the cost of (de-)activation is called the *break-even time* [17].

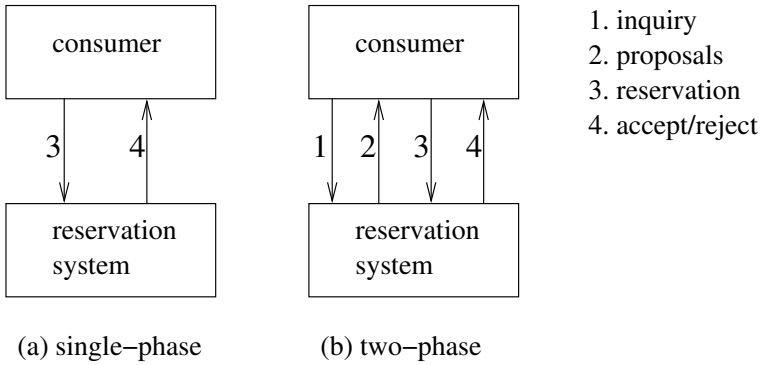


Figure 2.11. Two methods for reservation of resources.

The approach of exploiting deactivated modes can also be applied in entirely different contexts. The hard-disk example exploits component deactivated modes at run time. Devadas, Ashar, and Mauskar exploit deactivated modes at hardware circuit level (transistors) at chip *design* time [135]. By modifying the order of computations, they improve the power efficiency with about 40 %. For example, a dedicated processor needs to calculate $|a - b|$. The necessary operations are $a > b$, $a - b$, and $b - a$. The algorithms proposed by Devadas et al. create a dedicated processor that first calculates $a > b$, and, depending on the result, only activates hardware for either $a - b$ or $b - a$.

An approach to exploit deactivated modes with a higher level of sophistication is making the time-out period variable, called an *adaptive time-out*-based policy. This gives the policy the sophistication to adapt to the current usage pattern. Wilkes proposed this power management approach for hard-disks in 1992 (no simulations or other results are included) [187]. In 1995, Wilkes co-authored a paper with simulations of adaptive time-out policies for hard-disks [64]. They describe several methods to predict the duration of an idle period, such as using a low-pass filter, moving average, and conditional autocorrelation. In [51; 80], more algorithms and simulation results are presented. In [71] an overview is given of the research in this area and a general model for power management is presented.

Quality of Service

Instead of time-outs, a policy approach can be based on *Quality of Service* (QoS) and use concepts such as service negotiation and service contracts.

A large body of research around the topic of QoS has emerged; for an overview the reader is referred to [192]. A system that supports QoS is organized in such a way that components can provide a guaranteed level of service [41]. One of the key principles behind QoS is pre-determined resource allocation. Power management can be viewed as a special case of resource management and is therefore closely linked to QoS.

There is no common or formal definition of QoS. QoS research emerged from the network community. The International Telecommunication Union (ITU) standard X.902 refers to QoS as “A set of quality requirements on the collective behavior of one or more objects” [93].

Within QoS, reservations are used to allocate resources such as power. A policy that uses QoS knows and plans in advance the services it must deliver. Thus, a QoS policy has knowledge of the future. This fundamentally differs from a time-out-based policy that only has knowledge of the past.

Reservations for services are a key part of the QoS concept. Reservation systems for network

bandwidth are a topic of active research [200]. A reservation system can also be used for resources inside a portable device. An application sends a reservation request downwards and gets a response back. Most of the existing reservation solutions only respond to a reservation request with an accept or a reject. This implies that the application must downgrade the reservation to claim less resources and make another reservation attempt. This iterative negotiation process may slow down the system's responsiveness. An interesting approach, called NAFUR, is proposed in [73]. NAFUR provides what we call a *two-phase* reservation system with more feedback than a simple reject when a reservation cannot be granted. The two-phase and traditional single-phase reservation system are compared in Figure 2.11. Within the two-phase system, the first arrow indicates an inquiry for a service. The second arrow carries the feedback in the form of a list of *proposals*. The proposals are reservations that can currently be made that best fulfill the inquiry. Compared to the single-phase system, the two-phase system can make a satisfactory reservation with a fixed overhead of two information exchanges.

Several papers on resource management for QoS have been published, for example, the QoS broker [136]. A good overview of work in this area is given in [5].

Current QoS approaches often have a limited view of hardware. Within QoS research, the hardware is often regarded as a static entity that provides a *fixed* amount of system resources [197]. An important aspect in power management, however, is that hardware components may support multiple operational modes. For instance, hardware with an exponential utility curve operates very power efficiently at low performance. In our Perfd framework we take a broader view on hardware and explicitly take into account possible trade-offs that can be made by putting components in different operational modes (Chapter 4).

Benefit functions

A portable device that provides a poor, albeit, guaranteed service is still of little practical value to an end user. A policy approach that uses benefit functions tries to express the value to the user of each possible course of action. A *benefit-centric* policy maximizes the satisfaction to the user, while minimizing the power consumption. Such an approach holds the potential for a higher degree of power efficiency than approaches based on time-outs or QoS. However, user satisfaction is difficult to model and evaluating all courses of action in a benefit function is very computationally expensive.

Using functions to express benefit is described by Burns in 1991 in the context of hard real-time scheduling [29]. Burns describes a way to represent both hard, soft, and weak deadlines of tasks. Every task has its own benefit function $f : \mathbb{R} \rightarrow \mathbb{R}$ that specifies the received benefit $f(t)$ when the task completes at time t. For a task with weak deadlines, the benefit just becomes less when task completion is delayed. For tasks with a hard deadline, the benefit after the deadline either becomes 0 or even negative to indicate damage. The hard real-time scheduler uses the benefit functions to maximize the usefulness of the system. Benefit functions for processor allocation are related to the utility curves for hardware (Figure 2.2). The difference is that the utility curves state cost (power consumption), whereas benefit functions state reward.

The concept of a benefit function can be further generalized. The benefit can be dependent on a vector of parameters, instead of on a single parameter such as the completion time t. For example, the benefit function of video playback is dependent on parameters such as frame size, frame rate, and color depth [197]. Finding the optimal allocation in such large search spaces is NP-hard, and heuristics must be applied to find a suitable solution. For instance, simulated annealing was used to this purpose in [110]. Qu and Potknjak use benefit functions in a power management context to control a voltage scaling mechanism [154]. Their simulations show a saving of 39 % compared to using time-outs.

A general form of a benefit function is combined with QoS in [155] to form the widely used QoS-

Workload input	QoS service	Importance ranking	
		Priorities	Benefit
monitored	Best effort	Yes	–
given	Best effort	Yes	–
reservation	QoS guarantee	Yes	–
negotiated	QoS guarantee	Yes	Yes

Table 2.2. The four different types of input information concerning the workload.

based resource allocation model (Q-RAM). Q-RAM is an analytical model for resource management in systems with "multiple concurrent applications, each of which can operate at different levels of quality based on the system resources available to it" [155]. The goal of the model is to allocate the resources and maximize the system benefit. It is proven in [156] that this problem is NP-hard.

In [102] Kornegay, Qu, and Potkonjak propose to use benefit functions of applications to solve the clock scheduling problem. Their benefit function $B_n(c)$ shows the benefit of application n when allocated c CPU quanta. The benefit function is used to divide the finite CPU resource between applications while minimizing the required energy and maximizing the benefit. When applications specify their benefit function, the efficiency of the scheduling improves compared to the case where applications only provide a reservation. Reservations introduce uncertainties and inefficiencies because the difference between the worst-case execution time and best-case execution time may be quite large [57].

The benefit functions in [102] are synthetic. Benefit functions of real applications such as video decoding are complex. Even a simple model of video decoding with three parameters, frame size, color depth, and frame rate produces a hard computational scheduling problem [197]. To make matters worse, users may not notice a drop in frame rate, but still show stress [188]. In this interesting study users were shown video sequences with a variable frame rate of 5 or 25 frames per second (fps) while their physiological response was measured (heart rate, galvanic skin resistance, and blood volume pulse). In the study, 84 % of the participants did not notice it when the frame rate was lowered to 5 fps for 5 minutes, but their stress levels increased as indicated by their physiological response.

Using benefit functions means in essence communicating a range of operational modes and their respective benefit downwards towards the lower layers and leaving it up to the lower layers to select an operational mode. In a two-phase reservation system, the lower layers provide feedback on, for instance, the power consumption and resource usage for every operational mode. This enables the higher layers to select an operational mode. As a consequence, the benefit functions do not have to be communicated downwards. The higher layers only need to select the operational mode with the highest benefit and the lowest resource usage. Therefore, the difficult task of explicitly quantifying the benefit for each operational mode is avoided, as only the relative benefit needs to be known.

2.3.2 Input information

The input information to a policy has a significant impact on the power efficiency of that policy. The input information to a policy is divided into two parts (workload, operational conditions), according to their origin, as already depicted in Figure 2.6.

Workload

The input information concerning the workload has four known types. Each type is listed in Table 2.2.

The "monitored" workload type of input information consists only of past usage patterns or service requests. Extrapolations are needed to estimate the workload. Passive monitoring results in the lowest input richness. When input information contains future workload information, it is classified as the "given" type. This type of input information requires that the higher layer above the policy provides information about its future usage patterns or service requests. The "reservation" type is somewhat similar to the "given" type. The difference is the presence of feedback from the policy. When the policy provides explicit feedback with a commitment, the input is labeled as a reservation instead of mere given workload information. This sort of workload input information is commonly associated with policies that use a single-phase reservation system. A policy may offer a certain level of guarantee (QoS) for a reservation (Section 2.3.1). When a policy does more than simply approve or reject a reservation, the workload information is of the "negotiated" type. In this case, a policy receives a range of alternatives that fulfills the higher-layer needs.

All four types of input information can use a priority system to rank requests. For instance, monitored hard-disk requests are associated with a priority number. When high-priority requests are estimated to be somewhat likely to occur in the near future, the policy reacts different than when low-priority requests are somewhat likely.

Only the negotiated type of workload information may be associated with a level of user benefit (Section 2.3.1).

Having listed the four types of workload information, we will discuss several instances of workload-related input information such as deadlines, CPU needs, and future usage patterns.

Reservations of resources by the higher layers improve the efficiency of the CPU scheduling process and enable more accurate workload estimations. A general discussion on resource reservation issues can be found in [191]. In the area of real-time systems, such reservations enable the scheduler to provide hard guarantees on CPU availability. A reservation consists of the maximum needed CPU cycles and a deadline. The maximum needed CPU cycles can also be expressed as the Worst Case Execution Time (WCET). The classical Earliest Deadline First (EDF) scheduling algorithm [118] is optimal for real-time systems with a fixed processor frequency in the sense that all deadlines will be met if possible. Several papers have been published on clock scheduling for real-time systems [112; 146; 169; 170].

The required CPU cycles for a task must be estimated before this information can be shared with the lower layers in the system. For many applications or tasks, this is non-trivial [77]. For the task of decoding video frames several techniques have been investigated [15; 25; 131]. These estimation techniques for video require detailed knowledge on the format of the video bit stream.

Detailed information about the current workload has been shown to improve the power efficiency in some cases. With detailed workload information, extrapolation is more accurate and deactivation modes can be exploited more efficiently. The workload of a server component may be due to one or multiple concurrent clients. The identity of the clients is important when these clients have different usage patterns. In [121] a comparison is made between five hard-disk power management policies. Only one policy knows the client identity associated with each request. The other four policies must extrapolate the workload using only the aggregate workload. These policies are based on Markov models [185], regression relationship of idle periods [85], and advanced adaptive time-outs [97]. Their experiments show that the four policies without access to the client identity use 15 to 56 % more power. Therefore, information on the identity of the client is important for the lower layer that must schedule these requests and exploit deactivation modes.

The same authors of [121] argue in [122] that applications should communicate their usage pattern of devices to the lower layers of the system. Information on the future usage pattern of devices by applications provides more opportunities to exploit deactivation and to schedule service requests

Workload information	1998,[15]	1999,[102]	2000,[121]	2000,[122]	2000,[50]
client identity			simulations		
usage estimation	simulations				
usage pattern				prototype	
operational modes					simulations
benefit levels		proposed			

Table 2.3. Research publications with several examples of workload information for policies.

efficiently. In [122], an example with a text editor is given. A text editor that saves the current content to a file every five minutes (300 seconds) can use the following function to pass this usage pattern information to the lower layers: *UsagePattern(hard-disk,periodic,300000,500)*. The fourth parameter in this function indicates the maximum delay in ms that every save request can tolerate when re-scheduled.

In [50] an interface between components is proposed that exposes the operational modes of components. A benefit function goes even further [102]. Benefit functions expose both the operational modes and their value to the user.

Research publications discussed in this and previous sections are shown in Table 2.3. Listed are five examples of workload information, along with the relevant research publication. Also listed is the year of publication and the type of research results (prototype, simulations, etc.).

Operational conditions

The operational conditions in the layer below the policy, such as current WLAN conditions and battery level, can be used as input for the policy to improve power efficiency.

In many cases no information on the operational conditions is available to a policy. In other cases the current operational conditions can be extracted using statistics. For instance, current wireless link conditions can be estimated using packet transmission statistics. In this case the input information is labeled as "monitored". Another option is that operational conditions are given explicitly by the lower layers to the policy. This is called the "given" type.

Operational conditions are related to utility curves. A utility curve provides information on all possible trade-offs for a component. When operational conditions vary, the utility curve changes over time.

Several research publications discuss some instance of operational conditions information that is communicated from the lower layers up to the higher layers of the system. Operational conditions are very diverse. For example, the (wireless) network component can provide information on possible reservations to a video streaming application [73]. In [73] each possible (wireless) bandwidth reservation has different capacity, start time, and cost. A selection must be made between possible reservations such as a reservation for 1 Mbps that starts now for $1 and a reservation starting in 5 minutes at only $0.50 for the same 1 Mbps.

Informing the higher layers about the costs of their actions is an important source for evaluating alternatives. For example, during the design time of a dedicated chip, the power consumption is a metric in the synthesis process [34]. The power consumption of the current chip design must be communicated to the highest layer, which is the chip designer himself. A power estimation framework for software design time in the area of real-time systems is presented by Tiwari in 1994 [180] and further refined by Dick et al. in 2000 [49]. In order to design power optimized software, information

on the power consumption is indispensable.

Power consumption information can also be communicated at run time. In [197] a simple clock scheduling algorithm is proposed that knows the power consumption in each of the operational modes of a variable frequency CPU. When the CPU device driver communicates the performance/power settings upwards to the clock scheduling algorithm, it is possible to predict the remaining battery lifetime with a feedforward loop, instead of with the traditional feedback loop. The usage of CPU power consumption information for improving the battery lifetime prediction is proposed in [196]. They simulate the usage of a variable frequency processor with a program that uses the CPU and slows down the execution of other programs. In their experiments they obtain a remaining battery lifetime estimation from the OS using the standard Microsoft Windows GetSystemPowerStatus() function.

The current cost of a resource or a service is also an operational condition. Resource cost can be expressed in a general way, instead of in a specific unit of power consumption. An interesting economic-based approach using a virtual currency is proposed in [179]. The services of a component require the payment of this currency. The price tag for each service is communicated upward beforehand and applications pay this currency when the service is delivered. A similar economic-based approach is proposed for solving the cooperation problem in WLAN ad-hoc networks. The ad-hoc network relies on the cooperation of nodes by forwarding packets. Forwarding packets costs resources and some nodes may refuse to forward packets. The currency of "Nuglets" is used in [30] to pay for the relay of packets in a WLAN environment. Nuglets should ensure that nodes participate, because if they do not, they are not able to pay for their own bandwidth needs.

When a reservation system is used, the currently allocated reservations can be shared directly with the higher layers. This allows the higher layers to devise a new reservation that meets their needs and does not overbook the resource. In this case the explicit proposals in the two-phase reservation system are absent. A data structure for storing reservations that can be searched efficiently can be found in [163]. The process of finding the ideal fit of the application needs among the already allocated reservations is a complex problem. In [100] a "resource broker" is presented which solves this task on behalf of the applications. The resource broker is given information about the application's preferences and will make a reservation that yields the greatest application reward and is feasible given the already allocated reservations. The resource broker is given direct access to the currently allocated reservations.

Changes in the operational conditions trigger a signal to the higher layers in [37]. The current performance of the wireless link is communicated upwards in [104]. They propose to use this information to delay packet transmission for a few ms if channel conditions are poor, or to switch to an "error resilient mode" of the wireless link.

The remaining battery lifetime is important for many applications. This information is obtained at the hardware level by measuring the battery status [43]. This information is communicated upwards and available at the OS level in versions of Linux and Windows for access by applications. In an ad-hoc wireless network, the battery level can be used as input for the routing algorithm. Battery-operated devices with little remaining energy are not selected to forward network packets in [65].

A summary of this section is shown in Table 2.4. The table shows the five instances of operational conditions discussed in this section. Only the most prominent publication for each instance is included in the table.

2.3.3 Mechanism control resolution

Another property of a policy is the resolution of the commands that are used to steer the mechanism of a component.

Operational conditions instance	1997,[37]	1998,[73]	1999,[65]	1999,[179]	2001,[196]
power consumption					proposed
resource cost				proposed	
environment changes	prototype				
battery level			proposed		
possible reservations		proposed			

Table 2.4. A list of research publications investigating information exchange regarding operational conditions.

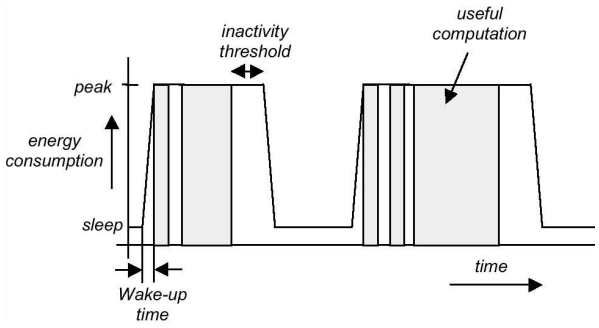


Figure 2.12. An example of binary mechanism control resolution, modified from [79]

A simple policy only produces on/off commands for the mechanism it controls. Sophisticated policies produce continuously changing parameter values for a number of commands. For example, a video streaming service where parameters such as frame rate, resolution, and color depth can be varied.

We distinguish four values for the control resolution property of a policy: binary, sleep levels, performance levels, and full setting. Each value indicates a different number of (de-)activated states and number of parameters that the policy controls.

- Binary, 1 deactivated state and 1 active state;
- Sleep level, multiple deactivated states and 1 active state;
- Performance level, multiple deactivated states and multiple active states;
- Full setting, multiple deactivated states and multiple active states with multiple independent parameters.

Policies with *binary* control resolution have been used extensively. The functionality of a policy is limited in this case to issuing only sleep/active commands. Figure 2.12 shows an example of a policy that issues such commands. These commands can be generated by a very basic policy, such as a static time-out-based policy (Section 2.3.1). Early hardware components only supported a single sleep mode. Components have evolved significantly since then, with hard-disks often taking the lead. Hard-disks are no longer sleep/active devices, but can be equipped with 5 power-down levels [86]. *Sleep level* control resolution is needed to control such components. Each sleep level (doze, sleep, hibernate, etc.) has a different wake-up time and power consumption. In addition to controlling one or several deactivated modes offered by the mechanism, a policy with *performance level* control

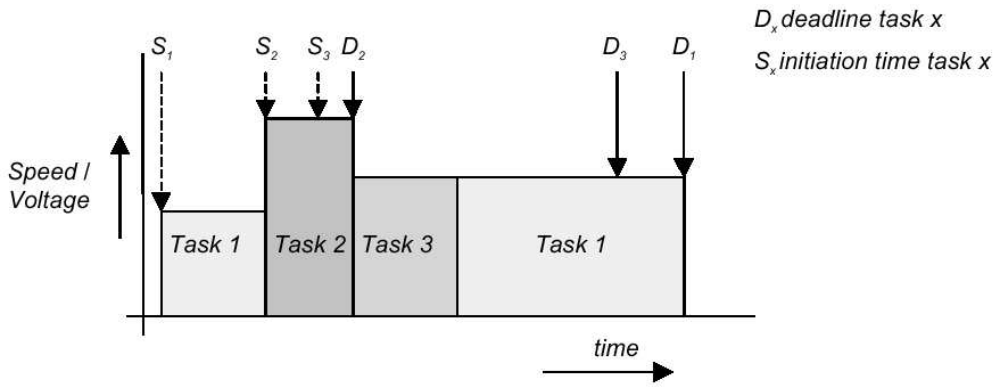


Figure 2.13. An example of a performance-level mechanism control resolution.

resolution also controls, for instance, the use of an economic mode and a high performance mode. For example, a general purpose processor can be set at various frequencies, each giving a different balance between execution speed and power efficiency (Section 2.1.2).

Weiser et al. presented the first simulations results of a policy that exploits the exponential utility curve of a processor [186]. Figure 2.13 shows the output of a policy that controls the processor. The figure describes the change over time of the processor speed/voltage and the task to be executed. Finally, *full setting* control resolution means that a policy has the sophistication to calculate several independent parameters of the mechanism. For example, it can be a policy that not only controls the speed of the processor, but also the size/speed of the cache [101], performance of the main memory, and current branch prediction strategy.

It is technically feasible to build mechanisms with parameterized sleep states. For instance, a fast activate and an economic activate for hard-disks. Due to the policy/mechanism interlock (explained at the start of this chapter), current hardware components do not support parameterized sleep states. As of 2003, proposed state-of-the-art policies in the literature are far away from being sophisticated enough to exploit them.

2.3.4 Time of fixation

Power management policies are fixed at some point in time and to a certain degree. The time and degree of fixation have a large impact on the power efficiency. For example, standards for power management, interconnection, and data exchange restrict a power management policy to some extent. A power management policy for a dedicated 802.11 WLAN chip equipped with the automatic rate selection feature (Section 2.1.3) is limited by the 802.11 standard in its possibilities to reduce power.

We distinguish four fixation times: standardization time, design time, compilation time, and execution time. A fixation time does not necessarily apply to the whole power management policy; it can also be limited to certain aspects of it, such as a timeout setting. The four elements of Figure 2.5 may have different fixation times. In the $|a - b|$ calculation example, (Section 2.3.1) the algorithm in the policy is fixed at design time, but the input information ($a > b$ or $a < b$) only becomes available during execution time.

At execution time, a policy must be fully determined. During standardization, design, and compilation a policy can be influenced to some degree. The Advanced Configuration and Power Interface (ACPI) specification [90], for example, already determines to some degree a power management policy. The ACPI standard is not generic, but biased towards power management policies with a static or

Time of fixation	1994,[114]	1995,[64]	1997,[99]	2000,[63]	2000,[84]
standardization			standard		
design	simulations			product	
compilation					simulations
execution		proposed			

Table 2.5. Research publications with different times of fixation for the policy.

adaptive time-out-based approach on a x86 PC architecture.

We now discuss several examples with different fixation times. We focus on the fixation of algorithms in a power management policy.

A prime example of policy algorithm fixation at standardization time is the 802.11 standard [99]. 802.11 defines a method to periodically deactivate the WLAN card (Section 2.1.3). The 802.11 standard makes it impossible for a WLAN card manufacturer to make a competitive product that both complies to the standard and includes a more efficient power management policy. A superior standard could use a second low-performance radio for periodic activation. An additional radio using frequency modulation could receive the Traffic Indication Map (TIM) field at roughly 1 Kbps at significantly less power consumption. An additional front-end with frequency modulation is relatively cheap. Unfortunately, the 802.11 standard only defines an inefficient periodic activation method.

The majority of the power management publications describe policy algorithms that are fixed at design time. The policy was fixed at *software* design time in the early research on static time-out-based policies for hard-disks [114]. In the Crusoe processor, the policy for clock scheduling is fixed at the *hardware* design time [63]. The Crusoe processor uses microcode on top of a VLIW architecture. The policy for clock scheduling is included in the hard-coded microinstructions. The policy measures the processor activity in fixed intervals and adjusts the processor frequency when needed. During execution time it is only possible to change the level of aggression of the policy or to disable the policy completely and set the processor at a fixed frequency. The ABLE policy for hard-disk power management (Section 2.4.1) is another example of a policy that is fixed at design time.

All components of Figure 2.5 are fixed at compile time in the simulations by Hsu et al. to solve the clock scheduling problem [84]. They propose to modify compilers such that these include power management commands in the produced binary code. The modified compiler detects memory-bound (nested) loops inside the source code and inserts a command to change the processor frequency before the start of the loop. According to their simulations it is possible to reduce the processor frequency within memory-bounded loops with negligible impact on performance. Unfortunately, the power saved within a memory-bounded loop may be insufficient to compensate for the cost of a frequency change.

Fixing the policy algorithm itself at execution time is difficult, because creating a complete policy algorithm at execution time implies automating the tasks of programming itself. In [64], a much simpler approach is suggested where several hard-disk policy algorithms are created at software design time. This simple approach selects a hard-disk policy at execution time, based on its quality history. When several policies are implemented, it is to some extent possible to compare their performance during execution. The performance of static and dynamic time-out-based policies for hard-disks can be calculated during execution without much overhead. To date, no publication has indicated whether such an approach may yield actual power savings.

A summary of this section is shown in Table 2.5.

Interaction model	policy input fixation	information flow
static	before execution	one-way
adaptive	during execution	one-way
cooperative	during execution	two-way

Table 2.6. The time of fixation and type of information flow for three different models of interaction.

2.3.5 Interaction model

The previous section classified the time at which (a part of) a policy is fixed. This section identifies three different models describing the interaction of a policy with the environment. The environment is defined as everything except the policy and the mechanism. The three models differ with respect to the time of fixation of the policy input and the type of information exchange with the environment.

Table 2.6 shows the three models of interaction. For each interaction model, the time of fixation is given and the type of information flow. A one-way flow of information indicates that the policy only consumes information. A two-way flow of information means that the policy not only consumes information, but also interacts with other policies, with lower layers, or higher layers.

A "static" interaction model means that the policy does not receive *any* information during execution time. An "adaptive" interaction model means the policy adjusts to changes in its environment based on input information at execution time. A "cooperative" interaction model means that the policy interacts with its environment. For example, a cooperating policy exposes its current resource reservations or cooperates with other policies to estimate the current workload.

The algorithms within a policy are only capable of interacting with their environment during execution time. Therefore, no interaction model exists for a two-way information flow at compilation time, design time, or standardization time.

2.4 Architectural properties

This section analyzes important differences between the supporting architectures for power management. Power management architectures differ in where policies are placed and how the interaction between policies is organized. The different values for the policy placement property and the organization property are examined in detail.

2.4.1 Policy placement

Power management policies reside at a certain level in the overall system. Policies can be located at roughly four levels: in the hardware, device driver, OS, or application. The location of the policy determines which information is available to the policy and how fast the policy can react to changes. Each location has its own distinct upper bound for power efficiency. We use two examples to demonstrate the effect of the location on a policy; hard-disk and voltage scaling policies.

The *end-to-end argument* is a classical principle in system design [161], saying that "functions placed at low levels of a system may be redundant or of little value when compared to the cost of providing them at that low level." Implementing functions in a higher layer of a system may yield a more complete, correct, and efficient solution. Examples in favor of the end-to-end argument include the correctness of a file transfer across a network. The correctness check should be conducted at the

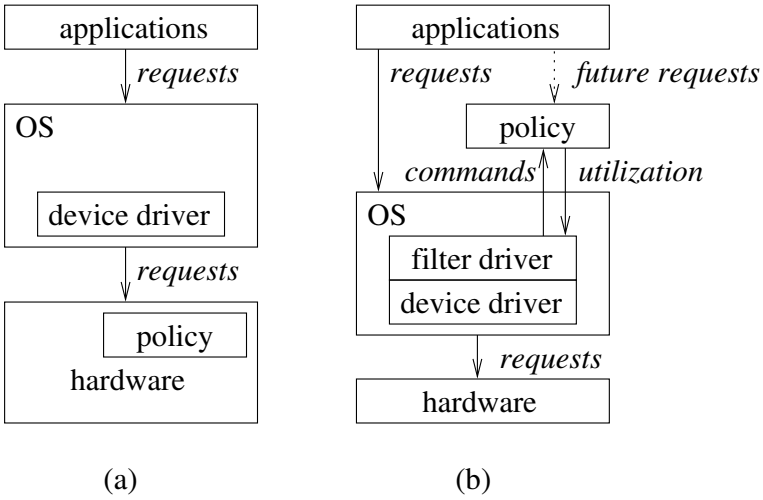


Figure 2.14. Two power management architectures with different policy placements.

highest possible layer in the end system, not at every intermediate network node. The same holds for encryption: end-to-end encryption by the application is desirable. The argument is also used in favor of the Reduced Instruction Set Computer (RISC) architecture. The instruction set should be as simple as possible, because any attempt to model the requirements of a whole range of applications is likely to be inaccurate. Better results are obtained when the instruction set is primitive and efficient.

If the end-to-end argument is also valid for placement of power management policies, it means that locating a policy at the application layer is superior. This superiority is consistent with the published results for specific cases, as we will shown for the clock scheduling problem (Chapter 5).

Hard-disk policies

Figure 2.14 shows two power management architectures. The architecture in Figure 2.14.a depicts a policy embedded in the hardware. In Figure 2.14.b the policy is located above the OS level and communicates directly with the applications. This is a policy located at the application level.

In [86] IBM engineers argue in favor of their "self-managed hard-disks" with an adaptive time-out-based policy inside the hardware. The IBM TravelStar hard-disk uses a policy in the firmware called Adaptive Battery Life Extender (ABLE). The architecture of Figure 2.14.a is used. No device driver or OS is involved in the hard-disk policy. The TravelStars series is equipped with five deactivated modes [86]. We will describe them all because they illustrate the need for a complex policy.

The TravelStars *performance idle* mode is entered immediately following the completion of requests, without any time-out delay. In performance idle mode, the servo motor keeps the platters spinning at full speed, but some of the electronics are powered down. Requests are processed with no delay. In *fast idle* mode, the power consumption is further reduced. The head is moved to a parking location, parked, and the servo motor control is deactivated. Wake-up time to active mode is about 40 ms. In *low-power idle* mode, the power consumption is reduced by 25 % compared to fast idle mode. The heads are unloaded from the disk, reducing power consumption. In low-power idle mode, the heads are not flying over the disk surfaces. Wake-up time to active mode is about 400 ms. In *standby* mode, the spindle motor is stopped, and most of the electronics are powered off. Wake-up

time is less than 2 seconds. *Sleep* mode is similar to standby mode, but uses less power.

The IBM ABLE policy knows the trade-offs for each deactivated mode. When no requests arrive for the hard-disk, ABLE puts the hard-disk in an increasingly deeper sleep mode. The increased power savings of a deeper sleep mode are continuously balanced against the larger wake-up penalty in both time and energy. The trade-offs are different for each TravelStar model; large-capacity disks use more power and larger disk caches also increase power. Thus, the ABLE software includes a *performance model* with break-even times [17] tuned for that specific model.

Because the ABLE policy is different for each TravelStar model, placement of the ABLE policy inside the device driver would result in a higher installation hurdle. Imagine that every IBM, Toshiba, or Maxtor hard-disk needs a CD-ROM or floppy with a specific device driver. Therefore, considerations beyond the pure technical arena play a crucial role in policy placement. Self-managed hard-disks with a standard interface such as IDE or SCSI require no device driver and still offer adequate power management.

However, according to the measurements published in [126], placing the hard-disk policy at the application level turns out to be even more power efficient. Lu et al. built a prototype of the architecture depicted in Figure 2.14.b for managing the power of an IBM hard-disk. Their adaptive algorithm saved as much as 25 % of the power.

Clock scheduling policies

A clock scheduling policy optimizes the processor frequency with respect to the workload to be serviced. In Section 1.4.1 we have defined the problem of clock scheduling. Section 2.1.2 provided details on the mechanism of voltage scaling.

In both [173] and [175], simulation results are presented on the performance of a clock scheduling policy put in hardware. In their technical report [173], So and Woo propose to use the path followed in the executed code to predict the duration of a task. The path begins after a processor wakeup and consists of basic blocks of code and follows branch instructions up to a certain depth. So and Woo propose to solve the problem of clock scheduling with similar techniques as those used in branch prediction [195]. This approach is likely to produce clock schedules with low power efficiency, because the processor hardware has no context awareness due to the lack of system-wide knowledge. Unfortunately, both publications on hardware-based policies do not compare their results with alternative policy placements.

Placing the clock scheduling policy at the OS device driver level is the approach of many publications. When it is placed at the OS device driver level, only processor load statistics are available from the OS task scheduler. Such processor load statistics are used to determine the clock schedule. The clock scheduler measures the processor load in fixed intervals (for example, every 20 ms). A simple technique is to adjust the processor frequency proportionally to the amount of processor load [186]. For example, if the processor was busy 15 out of the 20 ms in the previous interval; the processor frequency is reduced by 25 %. When there is no idle time in the previous interval(s), the processor frequency can be increased stepwise or exponentially. In 2000, Grunwald et al. published measurements on an implementation of a clock scheduler at the device driver level that uses the weighted average of the previous intervals to determine the new processor frequency [68]. Their measurements reveal “disappointing results”; the energy savings due to voltage scaling were only minimal, especially for bursty applications such as video decoding. Earlier simulations by Pering et al. in 1998 produced similar results [145]. In their simulations the power consumption of video decoding was 36 % above the optimum.

A clock scheduler at the device driver level has only access to information related to the device

Policy placement	1994,[186]	1998,[173]	2000,[68]	2001,[59]	2003,[153]
hardware		simulations			
device driver	simulations		prototype		
OS				simulations	
application					prototype

Table 2.7. Examples of research publications on differently located policies.

under control. By *integrating* the clock scheduler with the OS, other information becomes available to estimate the processing requirements of applications. Such "integrated clock schedulers" require numerous modifications to the OS. Flautner et al. [59] describe an integrated clock scheduler that maintains processor usage statistics of every process, observes the communication pattern between processes, keeps track of input/output device usage by processes, and tries to extract deadlines from periodic tasks.

The highest level at which a clock scheduling policy can be located is at the application level. In 2001, we measured the performance of an application-directed clock scheduler [153] (Chapter 5). An application-directed clock scheduler operates in close cooperation with the applications. Applications indicate their processing requirements and deadlines to the clock scheduler. Such cooperation between OS and applications is similar to a real-time OS. The difference is that the worst-case execution time is used in real-time OS scheduling, whereas in application-directed scheduling the average execution time is used.

For each of the four levels where policies can be located, example publications are shown in Table 2.7.

2.4.2 Organization

Power management solutions can be organized in different ways. This organization issue deals with the whole device, as opposed to dealing with only a specific component (Section 2.4.1, Policy placement).

There are two flavors for the organization of power management of a portable device: centralized and distributed. With the centralized option, a single policy manages the whole portable device. The distributed option has three approaches: two layers, three layers, or fully modularized.

In 1996, Udani and Smith proposed a centralized power manager, called the "power broker" [184]. This power broker will be "aware of the general state of the entire system and is in a position to make decisions that will enable the efficient use of resources". Figure 2.15 shows the structure of such a centralized power manager. The power broker contains the power management policies for all hardware components such as the display, hard-disk, CPU, memory, and WLAN. The power broker uses application groups to bundle applications with equal resource requirements and priorities. For example, groups for text editors, compilers, and web browsers. The power broker uses an "application registry" to store power management policies per application group. For example, during a telnet session the processor frequency is reduced, the hard-disk is powered down, and the WLAN is set to full performance. The basic assumption of the power broker is that only a single application is interacting with the user at a given point in time. With this assumption the power broker can simply determine the group of the current running application and use the policy associated with this group.

Centralizing the policies into a single component may yield good solutions in specific situations, but for the general use on a laptop or PDA it is impractical. The complexity of a central policy expands rapidly with the increase in supported hardware and applications. In the current PC-industry

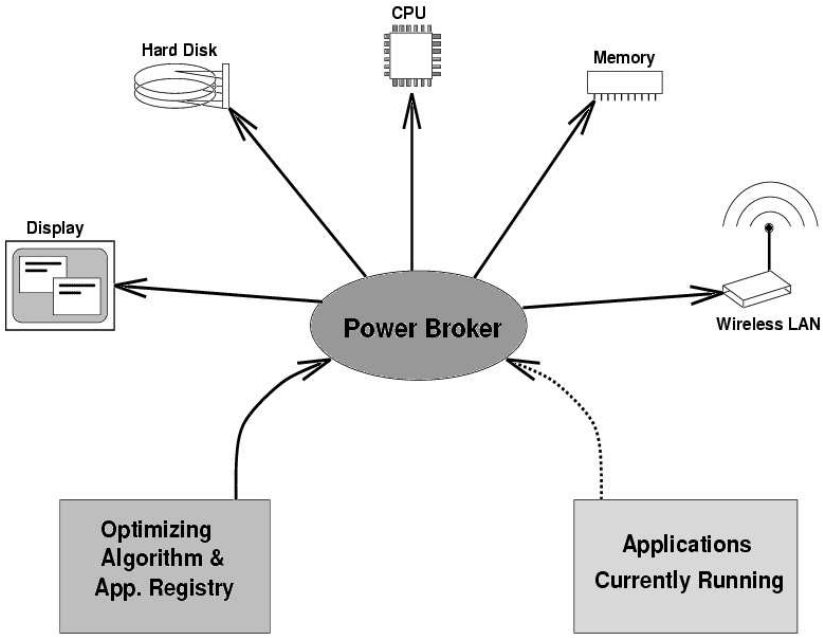


Figure 2.15. A centralized power manager called the power broker, from [184].

it is nearly impossible to create and maintain such a central policy. A single vendor would need to integrate the trade-off for the supported hardware and applications into a central policy. Therefore, we consider a central policy to be unscalable and hence unattractive.

Instead of a central approach it is also possible to use a distributed approach. The most simple distributed approach is the two-layer structure. The Advance Power Management (APM) [45] standard uses this simple approach. This standard is now outdated and replaced: Version 1.0 of this standard was released back in September 1993. APM has been developed exclusively for x86-based PC systems. APM places the policies side by side within the same layer; this differentiates it from the central approach. APM has some important drawbacks: policies are placed exclusively inside the PC BIOS and each hardware component is managed independently, making cooperation between policies impossible. The power management policies located inside the PC BIOS operate *without* the knowledge of the OS. APM defines an interface between the OS and the BIOS, for example, for informing the BIOS that the overall system is idle. The BIOS can also inform the OS of events, for instance that the battery is nearly empty. For specific devices such as the hard-disk a time-out setting inside the BIOS determines when it will be deactivated. In general, within APM the OS has no direct influence on the power management of specific devices and the sophistication of the policies is low, due to the limited space inside the BIOS for code.

A more advanced approach to power management is using three layers instead of two. ACPI [90] uses this approach. ACPI is developed to replace both APM and the Plug-and-Play hardware configuration standard. Compared with APM, ACPI moves the policies one level up. Within ACPI, the OS has complete control of the global system state and the state of individual devices. When the user presses the *suspend* button on his laptop, the OS is informed of this event through ACPI and may choose to delay the suspend action, for example, to complete a file transfer. Implementing the suspend action within the OS is more efficient than the APM solution in the BIOS. Using APM to suspend means saving the full content of the memory and full state information to the hard-disk, whereas

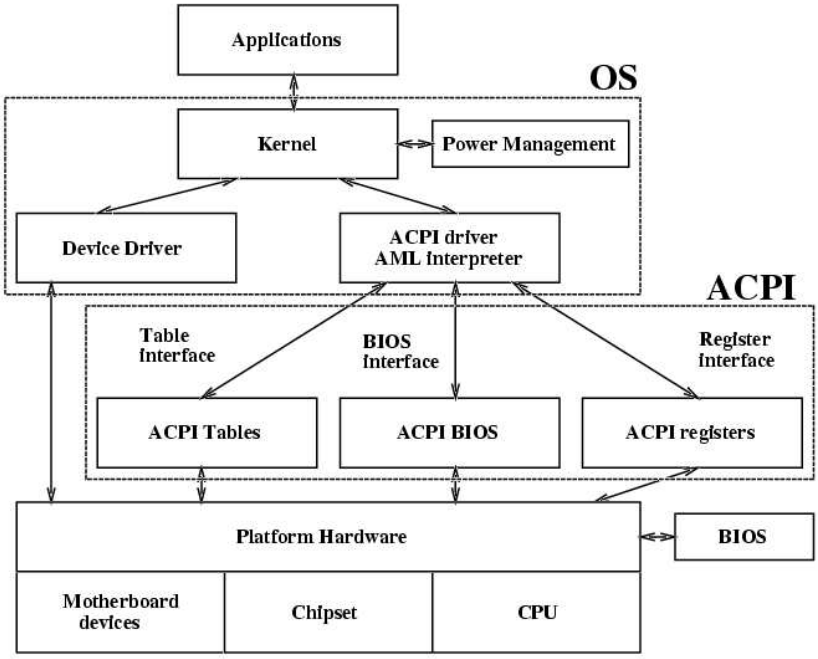


Figure 2.16. The placement of ACPI between the hardware and the OS, from [126].

in ACPI the OS only saves the memory content that is important to the hard-disk.

Figure 2.16 shows ACPI, along with the hardware and the OS. ACPI consists of three elements. The *ACPI BIOS* contains code to boot the device, implement the interface for sleep, wake-up, and some restart operations. The *ACPI tables* describe the hardware and contain functions for low-level operations in pseudo code. This pseudo code, known as ACPI Machine Language (AML), is interpreted by the OS using a virtual machine. A low-level function such as reading the temperature of the CPU is described in the ACPI tables using AML and can be interpreted and executed by the OS. The *ACPI registers* are the memory locations that provide an interface between the hardware and the OS for low-level configuration and power management.

A fully modular and de-centralized approach is proposed in [50]. Their approach is called Adaptive Resource Contracts (ARC) and provides an abstract view of a portable device. Policies are no longer confined to a single layer, but are fully distributed. Within ARC, the OS, device drivers, and hardware are abstracted and replaced by autonomous components that support QoS with a two-phase reservation system. The autonomous components are structured in a hierarchical fashion and together deliver useful services to the end user. Each of the autonomous components uses other components to deliver a required service. When one autonomous component requires a service, a QoS *contract* must be negotiated. ARC can exploit trade-offs between the different autonomous components to maximize power efficiency. Figure 2.17 shows the use of ARC in an application that delivers a live camera feed in compressed form across a wireless connection. Several contracts exist between the autonomous components in the figure. A contract contains constraints, for example, the minimal needed bit rate on the wireless connection. A contract may be violated, but violations must be signalled to allow other components to respond properly. ARC includes a concept similar to the configuration and operational modes in the Perfd framework (Section 1.4.1). A contract is created by selecting one of the operational modes of an autonomous component.

The research on ARC inspired the work on Perfd. ARC has a wider scope and is not specifically

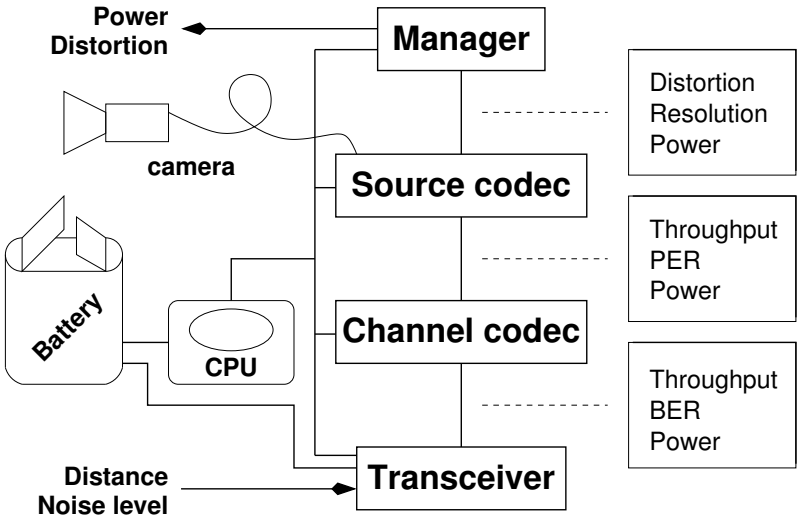


Figure 2.17. The distributed structure of Adaptive Resource Contracts (ARC), from [50].

Organization	1993,[45]	1996,[184]	1997,[139]	1999,[90]	2000,[50]
centralized		proposed			
two layers	standard				
three layers				standard	
modularized					simulations
hybrid			prototype		

Table 2.8. Examples of research publications with different organization of the supporting architecture for power management.

developed for power management. For example, ARC can also be applied to the management of a big team of researchers. A fundamental difference between ARC and Perfd is the level of abstraction. ARC abstracts the OS, etc. away and therefore maps less efficiently to portable devices that conform to the structure depicted in Figure 2.10. Due to the high level of abstraction in ARC, the resource usage during service delivery is not covered (the scheduling phase of Perfd)

Perfd uses a *hybrid* approach: it combines a central approach and a modularized approach. Perfd uses a single Perfd service plane at the center (Section 1.4.2). Unlike the central element of the power broker architecture, however, the Perfd service plane contains no intelligence and only coordinates the message passing. This hybrid approach is similar to the *Odyssey* system for managing the resources of adaptive network applications [139]. Odyssey uses a central entity called a *viceroy* to handle the intercommunication between various modules in the system. In [199] an OS extension is described that uses a central "credit manager" and a module for each hardware device. The credit manager is used to provide the guaranteed battery lifetime feature. In this enhanced OS, a credit must be obtained before a hardware device can be activated. The performance of the hardware is reduced to obtain a certain battery lifetime. Experiments showed that the power consumption of a laptop could be reduced from 3.5 W to 1.5 W. As a result, however, performance was also greatly reduced: the Netscape page loading time increased from 3 seconds to 29 seconds in their experiments.

Table 2.8 summarizes the various organizations of the supporting architecture for power management.

2.5 Taxonomy

This section presents the *Delft taxonomy* that classifies power management solutions.

We define a taxonomy as "a classification of concepts or entities in an ordered system that indicates natural relationships". As of 2003, no complete taxonomy for power management has yet been published. The Delft taxonomy for power management has been created to provide a starting point for a common vocabulary and to develop a framework that ties the many ad-hoc approaches in power management together. The aim is to identify all approaches for power management with a classification method that is relatively simple.

The Delft taxonomy provides a structured listing of all known approaches to power management. The next chapter of this thesis selects an approach with a high potential for power efficiency improvements, called the Perfd framework. The remaining chapters of this thesis are devoted to the implementation and evaluation of Perfd.

As the field of power management matures, insight is needed in the relative merits of the numerous possible approaches. Unfortunately the relative merits are difficult to quantize as, for example, it is difficult to quantify the costs of making applications context aware. A valuable contribution is the mathematical analysis given in [157]. This publication provides an *competitive analysis* [128] showing that a fixed time-out-based policy is inferior to a dynamic time-out-based policy.

2.5.1 Related work

In 1995 Golding et al. published the first "taxonomy of idle-time detection algorithms" [64]. The classified power management solutions in their paper are limited to monitoring the workload and exploiting the "off" mode; multiple sleep levels are not considered. Golding et al. present a general architecture of a policy and distinguish three elements: a *predictor* that monitors its environment, such as the arrival process of requests, and issues a stream of predictions, where each prediction is a tuple (start time, duration); a *skeptic* both filters and smoothes these predictions, possibly combining the results from several predictors; an *actuator* that obeys the sequence of predictions it gets as input to start and stop the device under control. For each of these three elements Golding et al. list the possible internal algorithms. For instance, the idle-period-duration predictor can be implemented using a fixed duration, moving average, back-off, or conditional autocorrelation. This initial taxonomy from 1995 is entirely focussed on the "solver algorithm type", it does not cover architectural decisions, cooperating policies, advanced mechanisms, and other policy input types besides monitoring. In 2001 a somewhat similar taxonomy was presented by Lu and Micheli [125]. They classified policies into three categories, based "on the methods to predict whether a device can sleep long enough". The three categories are time-out, predictive, and stochastic.

Lorch and Smith presented a very basic classification in 1998 [95]. They distinguish three energy management strategies. A *transition* strategy operates by constantly balancing the advantages and disadvantages of each mode a component can be in. This includes, for example, "how much power is saved by being in it, how much functionality is sacrificed by entering it, and how long it will take to return from it." A *load-change* strategy modifies the workload of a component in order to increase its use of low-power modes. For instance, "reordering service requests can reduce power consumption". An *adaptation* strategy aims to allow "components to be used in novel, power saving ways; an example is modifying file layout on secondary storage so that a magnetic disk can be replaced by low-power flash memory". This identification of different strategies by Lorch and Smith gives a classification on a higher level than that based on the solver algorithm type. However, the classification is too basic to serve as a basis for a common vocabulary to power management solutions. The definition of the

adaptation strategy could easily be re-labeled as *other strategies* and is too broad.

Instead of creating a taxonomy for power management as a whole, some authors present a taxonomy for only one aspect of power management, such as the adjustment to changing operational conditions or reservation systems/QoS.

The innovative and highly cited publication by Noble et al. on Odyssey contains a classification of approaches for "adaptation" [139]. Unfortunately, this publication does not include a definition of what characterizes an adaptive policy. The most stringent definition requires that a policy includes a self-correcting feedback loop that adapts the current behavior of the policy when previous behavior has proven to be too eager or too conservative. This example demonstrates the lack of a common vocabulary.

Noble et al. designed an architecture and implemented the Odyssey system that allows for the systematic monitoring of operational conditions and adjustment of applications to changing operational conditions [139]. Their classification is client/server oriented and identifies two important properties: fidelity and agility. *Fidelity* is defined as "the degree to which data presented at a client machine matches the reference copy at the server". *Agility* is defined as "the speed and accuracy with which it [the system] detects and responds to changes in resource availability". For applications they distinguish two extreme levels of application involvement for the adjustment to changing operational conditions. The first extreme, called *laissez-faire*, makes the adjustment entirely the responsibility of the individual applications. In the other extreme, called *application-transparent*, the system bears full responsibility for both adjustments and resource management. The middle road between these extremes, called the application-aware approach, creates a "collaborative partnership" between the system and the individual applications. In this case "the system monitors resource levels, notifies applications of relevant changes, and enforces resource allocation decisions".

In [7] a general model for adaptation to changing operational conditions is presented. This general model unifies the ad-hoc approach taken in six different systems (Odyssey, Transend, Conductor, Smiley, Coda, and Rutgers environment-aware API). Yet another classification for adaptation strategies is presented in [177]. This classification is strongly network oriented. Four different approaches are stated: a performance-based one, a feature-based one, a model-based approach using explicit network information, and a model-based approach using application benchmarks.

Taxonomies for reservation systems or QoS systems in general are the topic of several publications [5; 19; 160]. In [5] a generalized QoS framework is presented. This generalized framework includes 10 previously proposed QoS models, such as the ISO model, the IETF, model, and the TINA model. The fundamental differences between each of the QoS models are discussed. A number of properties are listed in this article for QoS management, QoS control, and QoS provisioning. The chosen values for each of these properties makes the 10 evaluated QoS models unique. The identification of QoS properties and their value range has a strong resemblance with the previous section that identifies and discusses power management properties.

2.5.2 Delft Taxonomy

The Delft taxonomy classifies power management solutions according to the properties of mechanisms, policies, and architectures as described in the previous sections of this chapter. Table 2.9 provides a summary of these properties and property values. For each property, such as the utility curve, a number of possible values proposed in the literature are listed. The Delft taxonomy simply uses these properties to identify power management solutions. To demonstrate its usage, we will classify several publications.

In 1994 an early and often-cited paper on hard-disk power management appeared [114]. The

Mechanism	Policy						Architecture	
Utility curve	Approach	Operational cond. input	Workload input inform.	Output resolution	Time of fixation	Interaction model	Policy placement	Organization
Flat Linear Exponential	Time-out QoS Benefit	None Monitor Given	Monitor Given Reservations Negotiated	Binary Sleep level Performance Full setting	Standard Design Compile Execution	Static Adaptive Cooperative	Hardware Device dr. OS Middleware Application	Centralized 2 layers 3 layers Modularized Hybrid

Table 2.9. Summary of properties and value for mechanism, policies, and architectures.

Publication	Description	Classification
APM 1993 [45]	standard for PC BIOS based time-out policies	Mx PTNGBSS AH2
Li 1994 [114]	simulation on a fixed timeout policy for a hard-disk	MF PTNMBDS Axx
Weiser 1994 [186]	policy to adjust the processor frequency to the workload	ME PQNMPDA ADx
Udani 1996 [184]	central "resource broker" that uses application classes	Mx PxNMPDA AMC
Noble 1997 [139]	architecture for adapting to variable conditions	Mx PQMRxDC AMH
Hafid 1998 [73]	framework for QoS negotiation with future reservations	Mx PQMNxDC AMx
ACPI 1999 [90]	standard for OS-based policies for x86 architecture	Mx PTNxPSA AO3
Lu 2000 [122]	a WLAN policy to exploit future workload indications	MF PxNGBDA ADx
Yuan 2001 [197]	adaptation based on changed resource cost and benefit	Mx PBMRPDC AMx
Zeng 2002 [199]	battery lifetime control through performance adaptation	Mx PQNMPDA AOH
Perfd 2003	Framework for cooperative power management	MX PXXNFDC AMH

Table 2.10. Several examples of the classification using the Delft taxonomy.

paper examines a policy that simply deactivates a hard-disk after a time-out period. Actual hard-disk usage traces were used to simulate the performance and power consumption of several time-out values. The power and number of hard-disk deactivations were simulated for time-out settings between 0 and 1000 seconds. This publication is focused on a single mechanism/policy combination; multiple mechanisms/policies and their organization are considered to be out of the scope of this paper. The paper considers only a single mechanism with a flat type of utility curve. The simulated policy monitors the workload in the simplest way possible and does not consider information about the operational conditions. The output resolution is binary; if the time-out period is exceeded, the sleep mode is invoked. The policy algorithm and parameter settings are fixed at design time and two-way interaction does not exist. The values for the properties listed in Table 2.9 are therefore $\{$*Flat, Time-out, None, Monitor, Binary, Design, Static*$\}$. The resulting classification is written in shorthand using only the first letter of each property value, with a grouping of mechanism, policy, and architecture property values. The first letter of the group is also included, either M, P, or A. The Delft taxonomy classification of [114] with respect to the mechanism and policy is written as "MF PTNMBDS".

A more advanced policy, which controls a WLAN, is proposed in [122]; it can improve power efficiency when applications indicate their future workload. The policy uses the given future workload information to exploit sleep modes more efficiently. The interaction between policy and application is only one way, the policy does not provide any QoS-like guarantees. The type of utility curve for the mechanism is not included. The proposed policy is located in the device driver. Because the value for the architectural property "organization" is not described, the value "x" is used instead. An "X" denotes that all possible property values of Table 2.9 are explicitly mentioned and can all be used. This yields the following classification "Mx PxNGBDA ADx".

Table 2.10 shows a number of publications classified according to the Delft taxonomy. Each publication has been previously mentioned in this chapter. For each publication we give the first author, year of publication, bibliography reference, description, and classification.

The increasing sophistication of power management becomes clear from this table. As time progresses, publications that increase the efficiency by exploiting sleep modes become rare. With the APM and ACPI standards, time-out-based policies are mature, further improvements are difficult and offer little power-efficiency gain. Cooperative power management solutions that involve applications in power management and exploit two-way communication at execution time are beginning to emerge in publications from 1997 onwards. The number of publications on cooperative power management

is still limited and advances are difficult. However, there is still room for a significant gain in power-efficiency.

2.6 Conclusions

Our discussion on the major concepts in power management ended with the first complete power management taxonomy. The proposed Delft taxonomy is by no means the final answer. Innovative power management solutions may introduce new properties and new property values that were not considered. Properties may need to be replaced to remain abreast with the maturing field of power management. Other proposed taxonomies in related work are very basic. The Delft taxonomy strikes a different balance: it is developed to be detailed enough to be useful and simple enough to be generic.

A range of alternative classifications are possible for power management solutions. The listed properties are in our opinion the most influential properties of a power management solution. The classification method does not use properties with just two values such as *absent* and *present*. We believe that such a binary property does not represent a useful and fundamental characteristic of a power management solution. By definition it is not possible to extend a binary property with a new value. The property *Approach* can be extended with new and novel solutions, the binary property *uses QoS* cannot.

The Delft taxonomy provides valuable insight in the progress of power management solutions. It helps to identify mature solutions such as time-out-based policies and emerging solutions such as cooperative power management. The cooperative Perfd power management solution proposed in the next chapter can hopefully serve as a foundation for the next standard in power management.

Chapter 3

Perfd framework

THIS chapter analyzes the problems and opportunities concerning power management in portable devices centered around a general-purpose CPU. As a result we have designed a power management framework for these devices, called Perfd. The Perfd framework has been specifically designed to exploit the opportunities offered by context-aware policies and increased information richness.

We explain the inner workings of Perfd in detail. Perfd has been implemented on an embedded platform for experimental validation. This chapter describes this platform and our measurement environment. This environment enabled us to evaluate the ideas within the Perfd framework using actual performance and power consumption measurements.

Roadmap

3.1 Problems and opportunities

This section revisits the power-management problem and all possible solutions discussed in the previous chapter. The central research question is: what direction in power management holds the highest

potential to advance the field of power management? The answer to this question has been used to guide the design of Perfd itself.

3.1.1 Observations

Most of the advanced solutions for power management listed in the previous chapter are only found in research publications. This section provides a short description of existing power-management solutions in commercial products as background knowledge for the subsequent discussion on research challenges.

As of 2003, power management in commercial portable devices is almost entirely based on the *time-out* approach. For instance, devices such as portable PCs using a flavor of Windows, small PDAs running Linux, and smart phones running Symbian all use time-outs for powering down the screen, processor, and hard disk. The only exceptions are some types of laptops with a simple policy to exploit voltage scaling. The input information for commercial power-management policies is mostly limited to monitored workloads and the output resolution is limited to binary or sleep-level modes. We estimate that currently at least 95 % of the different policies on commercial portable devices falls within this class (yielding a "PTNMSDA" classification, Section 2.5.2). An increasing, but still small, percentage of laptops monitors the CPU activity to estimate the minimal required frequency (the most simple form of clock scheduling, yielding a "ME PxNMPDA" classification).

The architecture of portable devices with a general-purpose OS has not been significantly enhanced over the years. The five-layer architecture (hardware, device driver, OS, applications, user; see Figure 2.10) is firmly embedded in the design of portable devices, despite all the published research on the topic of middleware. The latest version of Windows (XP) has a basic infrastructure for the registration of multimedia decoders and encoders. Such an initial step towards full modularization is still lacking in Linux. Exchange of non-functional information between layers is very rare. Interfaces only pass the bare essential functional information to maximize the level of transparency. Input richness of power-management policies generally suffers as a result.

A very important consideration obstructing innovation in power management is its financial implications; often, an unrealistic trade-off is made. This trade-off is the cost of implementing a power-management solution versus the improvement in power efficiency. The holy grail for many power-management researchers is to improve the power efficiency without *any* modification to applications [13; 59] or software in general [63]. It is questionable whether a "magical recipe" exists that will significantly improve power efficiency *without* modification of software. Therefore, the future of power management in commercial portable devices will lie somewhere between the two extremes, no software modification and minimal power efficiency improvement or extensive modification and maximal improvement.

3.1.2 Research challenges

To identify the power-management research challenges, we first determine the requirements for the next generation of power management schemes. In our view, the ultimate goal in the broader context of mobile computing is *making portable devices more useful and intelligent*. We identify four research challenges that must be tackled in order to reach this goal, being context awareness, modularization, information access, and pro-activity. Consider our scenario that describes the hypothetical "Aware" system, inspired by the Aura system [162].

> Fred is in a meeting, he just gave a presentation and is very tired. He grabs his Palm XXII wireless hand-held computer. Aware transfers the state of his work from the

> presentation desktop to his hand-held, and allows him to make some corrections to his presentation. After his meeting Fred travels to the train station to go to the next appointment. Fred wants to relax and selects the Aware Personal MTV service on his hand-held to watch some video clips. Aware infers where Fred is going from his calendar and the location tracking service. The personal MTV service sees that bandwidth will be scarce for the next hour. Some interesting video clips are pre-fetched at the train station with abundant wireless bandwidth. The personal MTV service calculates the battery lifetime for numerous service settings and selects almost the highest video playback quality with the highest screen illumination since it knows that there are conveniently located power outlets in the meeting room of his next appointment. When Fred arrives at his meeting the hand-held battery is almost completely empty and Aware asks Fred to feed his handheld.

To realize the above scenario, software in general must have full understanding of its environment. Context awareness should become an *integral part* of a portable device. At one end of the spectrum, the *end* user's context and intent [165] must be understood by a system such as Aware. At the other end of the spectrum, the power-management solution on the Palm XXII should add another type of context awareness, such as the costs of resource usage. Research that merely increases the efficiency of exploiting deactivated modes does not bring us significantly closer to our aim of more intelligence in portable devices.

A formidable challenge is moving the whole field of power management away from time-out-based solutions towards more context-aware solutions. Power-management policies must evolve to understand and reason with concepts such as workload, performance, quality, operational conditions, and power consumption. It is very likely that in this respect changes must be made to the OS and applications.

Monolithic software applications are still the rule rather than the exception; modularization of software is still in its infancy. The advantage of modularization from the power management perspective is the possibility for fine-grained control by using distributed power-management policies (Section 2.4.2). A great need exists for an open framework with re-usable software modules. However, creating such a framework is a huge task (e.g. CORBA, JavaBeans). First, the basic infrastructure needs to be created for module loading, registration, etc. Second, the labor-intensive development of numerous standard modules. Examples of standard modules are multimedia decoders, web browsers, position calculators, and speech recognizers. Therefore, modularization is a significant challenge that is likely to take time to evolve. The challenge is to accommodate a gradual transition from current monolithic applications towards context-aware modules.

Power-management policies that have the sophistication to understand their environment are useless when they do not have access to information about their environment. Currently, components of portable devices only share functional information. The next generation of power management needs to facilitate the sharing of more than functional information. The challenge will be to define the appropriate interfaces for sharing this information. Independently developed components need a common interface. For example, a standard API needs to be defined for the CPU that allows applications to calculate the costs of processing and communicate their processing needs, independent of the processor architecture, model, and speed. Such interfaces are a challenge on their own, because they need to be specific enough to improve power efficiency, yet generic enough to be useful for a whole range of components.

The final and most difficult challenge is adding pro-activity to portable devices. Current state-of-the-art portable devices are still mostly limited to passive behavior; portable devices have no clue about the user's intent [162]. For instance, a user starts watching a DVD movie on a laptop and the

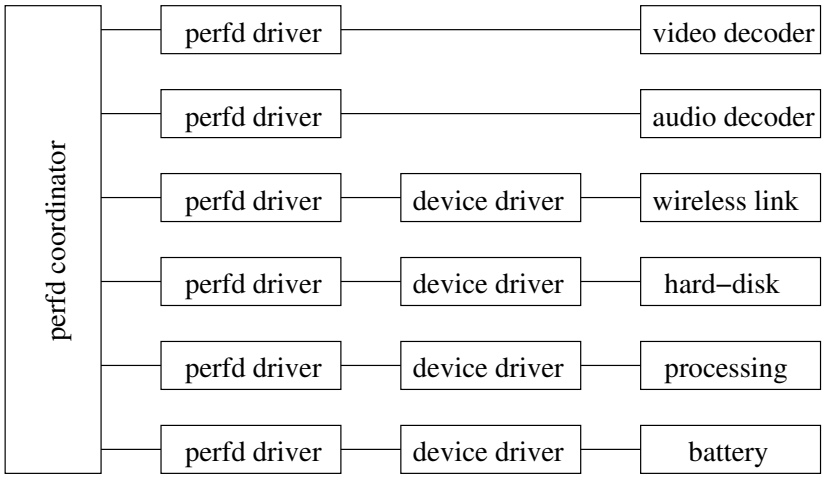


Figure 3.1. The structure of the Perfd framework for a multimedia terminal.

remaining battery lifetime is insufficient to see the whole movie. Current power-management policies are not pro-active and do not have the foresight to predict *and influence* the remaining battery lifetime. Pro-active behavior is a key element for useful and intelligent portable devices. The challenge is to design a power-management solution that also facilitates pro-active behavior.

3.1.3 Approach

The above challenges can be met by applying cooperating policies. Recall that cooperating policies (Section 1.4.1) are power-management policies that understand their environment, interact with each other, exchange non-functional information, and can estimate/influence the future. The potential of cooperating policies has been discussed before [16; 17; 55; 139]. However, an architecture to support cooperating policies has been lacking. This is best illustrated by the state of affairs in the the area of reservations. Reservations are a powerful instrument for cooperating policies to inform others of future events. As observed in [193] "a higher-level framework" is missing to coordinate reservations. As of 2003, only isolated solutions exist for different components such as Rialto's CPU reservations [96], RSVP network bandwidth reservations [200], and hard-disk access reservations [22].

Our solution to power management is based on numerous ideas. Most of the ideas are inspired by research publications outside the power-management field on topics such as modularization [44], performance modeling [82], and QoS [5]. Several other research publications in power management exist that present a combination of ideas [139; 184]. The key difference is that our combination is based on feedback from initial implementations and power measurements.

Our Perfd framework supports a modularized type of organization as listed in Table 2.8. We define a component as something that delivers a non-trivial service and uses resources. Perfd also supports a gradual evolution from the current dominant three-layer model into a modularized type of organization. A key feature of Perfd is its decentralized power-management policy. A central "resource broker" model has many drawbacks such as poor scalability. Creating decentralized policies is difficult because the different policies must work together, but by definition lack a single controller. The policies also need to deal with the limitations imposed by general-purpose OSes. For instance, OS device drivers are not abstracted away in the Perfd approach. Figure 3.1 shows the architecture of Perfd from an implementation viewpoint. The right-hand side of the picture shows the different

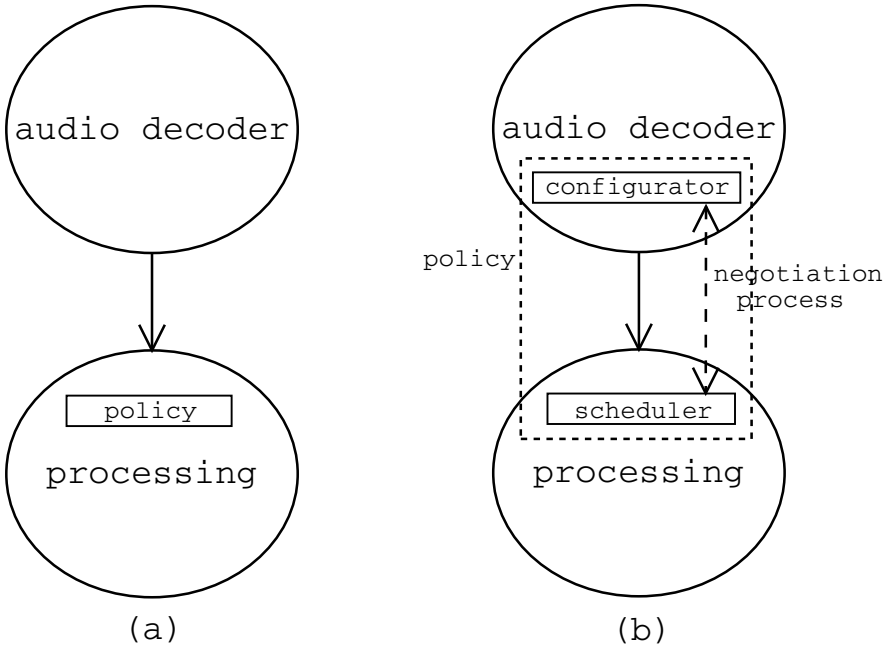


Figure 3.2. Perfd components with a traditional policy (a) and a distributed policy (b).

hardware and software components. Together a hardware component and a device driver form a power-management mechanism that must be controlled. The decentralized policies are located inside the Perfd drivers. No central intelligence exists in the Perfd approach; the Perfd coordinator merely passes messages around and stores shared data structures.

The central, new concept in Perfd is adding the ability to communicate and reason about alternatives. These alternatives can exist either at a high level, such as the level of compression used within a service, or at a low level, such as which task ordering for the next second yields the lowest processing power consumption. The communication about alternatives is explicitly supported by the Perfd coordinator. This is one aspect of the Perfd framework addressing the key research challenge of increased access to information. Reasoning about alternatives implies having decision points and the ability to predict the likely outcome of alternatives. Such outcomes are predicted using *performance models* at execution time. Performance models provide the ability to quantify the power consumption of alternatives at the cost of some run-time overhead and additional software development at design time. The performance models are situated in the Perfd drivers (Figure 3.1). The Perfd framework provides basic support for pro-active behaviour. Pro-active behaviour is supported by the performance models that can estimate the future and the ability to reason about alternatives.

Figure 2.1 showed the traditional model of combining a policy and a mechanism. This basic model does not explicitly support pro-active behavior. When multiple components are combined, the most simple solution is to simply stack them together in a hierarchical form (Figure 2.8). However, in this simple model, the intelligence (policy) is included in one component. The efficiency of this simple model can be improved by splitting the policy into two parts. We call these parts the scheduler and the configurator (see Figure 3.2b). This split causes a change in the communication model between a client component and a server component. The traditional model changes from a demand for service into a cooperative model with a negotiation process between the scheduler and configurator in which alternatives are explicitly exchanged.

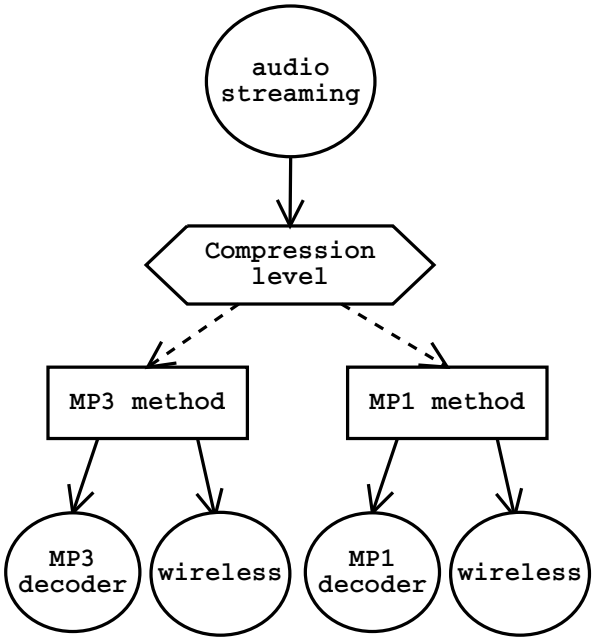


Figure 3.3. A config-space graph of the audio streaming service example.

Figure 3.2 shows both the traditional model with a monolithic power-management policy (a) and the distributed policy model of the Perfd framework (b). In the traditional model the audio decoder simply uses the processing component without any knowledge of resource usage and power consumption. In the distributed policy model, the configurator communicates its requirements and the scheduler deals with the limits. Through cooperation they can find a superior solution.

The scheduler shields the upper component from the details of its underlying mechanism. The configurator negotiates the most power-efficient operational mode with the scheduler and internally makes alternatives explicit. The scheduler determines for each service request of (multiple) components the most power-efficient access pattern for the mechanism it controls. Components can now cooperate to find the most power-efficient solution. So when a cooperating policy (configurator, scheduler) is split, both components contain intelligence and context awareness.

An "audio streaming service" is used as a leading example in the remainder of this chapter. Chapter 4 describes several measurements on the implementation of this service. The audio streaming service uses a wireless link (802.11) to stream compressed audio data from a multimedia server to a portable device. On this portable device the audio data is decoded by the processing component.

The cooperation between the scheduler and the configurator is best illustrated with the Perfd concept called "graph search". For complex services that involve multiple components, such as the audio streaming service, it is non-trivial to determine the optimal operational mode for each component. The dependency of the audio decoder and the processing component is shown in Figure 3.2; we call this a *dependency graph*. A *config-space graph* enhances a dependency graph with alternative configurations for the realization of a service. Figure 3.3 shows a simplified config-space graph of the audio streaming service. The dotted lines indicate the existence of several alternatives. A config-space graph is used during execution time by the various Perfd configurators in a complex service to make decisions explicit and to calculate the resource usage for each alternative course of action.

Alternative configurations use different operational modes for one or more components. For instance, for audio streaming the MP3 format implies heavy decompression and the MP1 format light

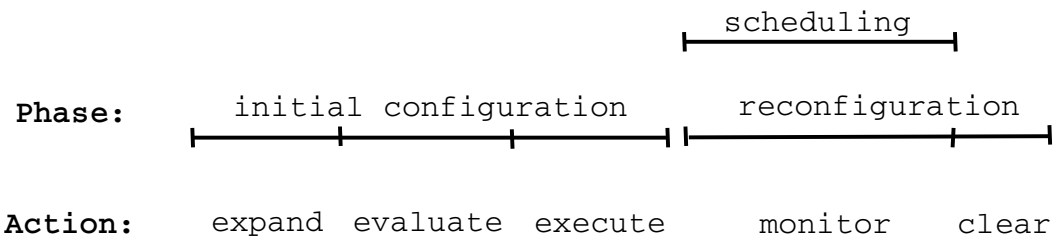


Figure 3.4. The different actions during configuration and scheduling in Perfd.

decompression. The processing element is omitted in the figure for brevity. "Compression level" is the name of an explicit decision point at execution time. Decision points improve power efficiency in this particular scenario by selecting a certain compression level. In the traditional approach the compression level would have been fixed at software design time, independent of factors such as the hardware platform or wireless link conditions. The explicit naming and evaluation of alternatives using a performance model is new to the best of our knowledge; Perfd is the first power-management framework to have this ability.

The delivery of a service using a configurator and scheduler is divided into several actions (expand, evaluate, execute, monitor, clear). Figure 3.4 shows the name of the phase prior to service delivery (initial configuration) and during service delivery (both scheduling and reconfiguration). During the expand action, a config-space graph is constructed. The evaluation action of the various components involved adds concrete power consumption information to the config-space graph using performance models. Execution means starting the service. During the monitoring action, the schedulers are active to ensure that the optimal operational mode is maintained and the configurators are active to check whether the configuration is still optimal. The clearing action handles the deactivation of the service and initiates sleep modes if possible. Each of these actions will be explained in detail in the next section.

3.2 Architecture

This section explains the details behind the Perfd architecture and the underlying design rationales.

3.2.1 Design rationales

The design of the Perfd architecture is based on four design rationales.

1. Enable modularization of components.
2. No difference between hardware and software components.
3. Trust other components, no QoS contract enforcement.
4. Seperation of the different component functions, no integration.

The first Perfd design rationale is enabling modularization. Breaking up applications into flexible, smaller, and re-usable components is the subject of numerous publications. The Perfd framework is closely related to the field of middleware. Middleware researchers also aim to enable modularization, for example, with standards such as CORBA [44], DCOM [24; 39], and JavaBeans [133]. From the

numerous middleware publications, only a few focus on portable devices. An architecture is needed to support components on a portable device. Proposed portable device architectures all suffer from drawbacks. For instance, the Odyssey architecture proposal to support components is in our opinion the best attempt published to date [139] (Section 2.5.1). Drawbacks of Odyssey are the focus on quality instead of quality/cost, the fixation on network bandwidth, and the limited interface between components (based on the file analogy).

The second Perfd design rationale is that hardware and software must be treated equally. This design rationale aims to remove the existing artificial boundary between hardware and software. Component architectures seldomly provide equal treatment to the components below the OS, compared to the components above the OS. Numerous architectures do not even consider the lower components at all. Providing a unified view of hardware and software has been hinted at before in this context, but was never realized [197]. Providing a single abstraction for certain service classes or hardware device types is suggested in a number of publications [139; 184; 72]. However, defining interfaces has proven to be a difficult task. For example, the interface for the processor is the topic of a considerable number of publications [145; 112; 116]. Yet, despite the large number of publications, there is still no commonly accepted processor component interface.

The first two design rationales have a significant impact on the interfaces between components. The use of the component paradigm almost automatically implies the fixation of interfaces, as interfaces must be pre-defined when applications (re-)use existing components. The alternative is not fixing interfaces and letting the various component developers define them. This approach without standardization will most likely result in numerous incompatible interfaces and hamper component development. In Perfd, components must belong to a certain type and have a pre-defined interface. For example, a video decoder component to be used may only be passed in a specified way information such as a filename, a compressed video stream, or a query for a resource usage estimation. For generic component types, such as storage, multimedia decoding, and web browsing, it is possible to define generic interfaces. For instance, the USB standard defines a generic interface for storage components using diverse technologies such as magnetic disks, optical disks, and FLASH. Of course, flexibility is an important concern when interfaces are fixed. Subtypes and interface versions are used in the Perfd architecture to ensure that the developer of each component has the freedom to balance efficiency, compatibility, and economical concerns. Architectural problems resulting in, for example, "DLL-Hell" [134] (Windows specific problems caused by the lack of sufficient support for different library versions) have limited the use of the component paradigm mostly to pre-packaged libraries instead of the more ambitious middleware approach of CORBA and DCOM [48].

Using modularization implies clustering of the functionality of a portable device into distinct and independent components. For example, in a wireless multimedia terminal, components can be clustered into a display, wireless link, battery, video decoder, and processor. Clustering is not a trivial task because efficiency is a major concern. It may seem natural to separate the CPU and memory into two different components, but due to the significant entanglement between them this is not desired; software will always require both the CPU and memory. A more efficient approach is to merge them into a single *processing component*. This component is a wrapper around the CPU, various cache levels, and main memory.

The third Perfd design rationale states that components in a portable device trust each other. *No* resources are spent in Perfd on "QoS contract enforcement" [42]. This design rationale is motivated by the fact that there is only a single end user of a portable device. In [198; 199] the idea of using "currentcy" (short for current and currency) was presented to implement a feature that we call "determined battery lifetime". In the currentcy experiments on a Linux laptop, the authors delayed hard-disk requests and slowed down software execution to meet a given battery lifetime. Techniques

such as budgets, debt limits etc. were used to keep track of every instance of power usage at a very low level (1 unit of currentcy equals 0.01 mJ). In the Perfd framework we carefully avoid such accounting overhead by implying trust on components not to waste any energy.

The fourth Perfd design rationale states that the different logical functions of a component must not be integrated. In the publications on middleware and adaptivity research, little attention [13; 179; 191] or no attention at all [72; 95; 139] is given to a general architecture of a component or a power-management policy itself. There is no commonly accepted detailed approach to structure a component. A general design guideline exists that promotes separating a power-management mechanism from the controlling policy within a component. A lesser known guideline called the *separation principle* is described in [113]. This publication in the area of architectures for adaptivity advocates the separation of measurement and control functionality. For the area of power management, another observation is important. No commonly accepted approach exists to structure a power-management policy.

Creating a power-efficient cooperating policy is difficult without any guiding principle. The fourth design rationale explicitly rejects monolithic policies. Solving smaller clearly defined problems is simpler than solving a large problem. Therefore we split a component into several parts. We have identified five logical functions that must be present in a cooperating policy. Together with the power management mechanisms (6th) these logical functions form a single component. When such logical functions are explicitly identified and exploited in a policy, both the development process and power efficiency is likely to improve. The logical functions within a component are thus as follows.

- Scheduler
- Configurator
- Workload estimator
- Operational conditions estimator
- Performance model
- Component services

Component services is a generic term to denote the parts of a component that implement the power-management mechanism *and* the services of the component.

3.2.2 Component model

Figure 3.5 shows the abstract component model that includes all logical functions. Several of such components joined together realize services for the end user such as the audio streaming service.

The *scheduler* determines the most power-efficient access pattern for the component services. The top-most components, applications, do not require such a scheduler. Hardware components can often only service a single consumer at one point in time. The scheduler can use time slots to grant exclusive access of service consumers to hardware.

The *workload estimator* inside the scheduler uses different sources of information to give an estimate of current and future requirements. Sources of information include previous requests, reservations, and other types of information.

The *component services* offers the actual functionality of the component, either hardware or software based. The component services also include the power-management mechanism controlled by

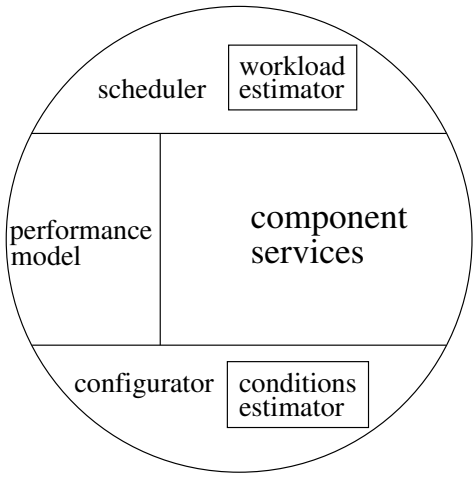


Figure 3.5. Abstract model of a component within the Perfd framework.

the scheduler. Component services examples are a software-based video decoder and the hardware of a disk.

The *performance model* of a component contains detailed information on quality and cost trade-offs in using that component. This performance model can be queried to provide a power consumption estimate before any service is delivered. This model includes the specific characteristics of each supported operational mode.

The *configurator* selects the most power-efficient operational mode for the required underlying services for that component. For example, a component using a wireless link needs to balance bandwidth and power consumption. The configurator/scheduler relation was already shown in Figure 3.2. Each configurator connects to a scheduler of an underlying component. Both cooperate to find the most power-efficient configuration and access schedule. The configuration part of a policy is responsible for both setting and guarding the optimal setting of the underlying services. Figure 3.6 shows the dependency graph of all components involved in our audio streaming service example. Setting and guarding the optimal setting is not trivial because multiple components are involved, each having a different perception of time. For instance, 802.11 wireless LAN packets are transmitted every ms, while pre-fetching and caching actions involving a hard disk may occur once every few minutes.

The selection process of operational modes is a vital step in devising the most power-efficient configuration of a portable device as a whole. This configuration step is carried out before the delivery of a service. The quality and cost of the underlying service are monitored during service delivery by a special *operational conditions estimator* within the configurator. The configurator uses this quality/cost information to consider the need for a reconfiguration step. Such a reconfiguration step means halting service delivery, changing parameters, and resuming service delivery.

Again there is no commonly agreed terminology for most of the above concepts. In several publications a dedicated element is responsible for the estimation of the workload or simple logging of incoming service requests. This element is called a “filter driver” in [13], “episode detection” in [59], and remains nameless in [121]. The lack of a commonly agreed terminology indicates the immaturity of the field and the need for a unifying framework such as Perfd.

Figure 3.7 shows a more detailed component model that includes the information flow. The scheduler receives several types of information. Incoming reservations (two-way information exchange, Section 2.3.5) are processed by the scheduler and result in either an accept or a reject. The usage

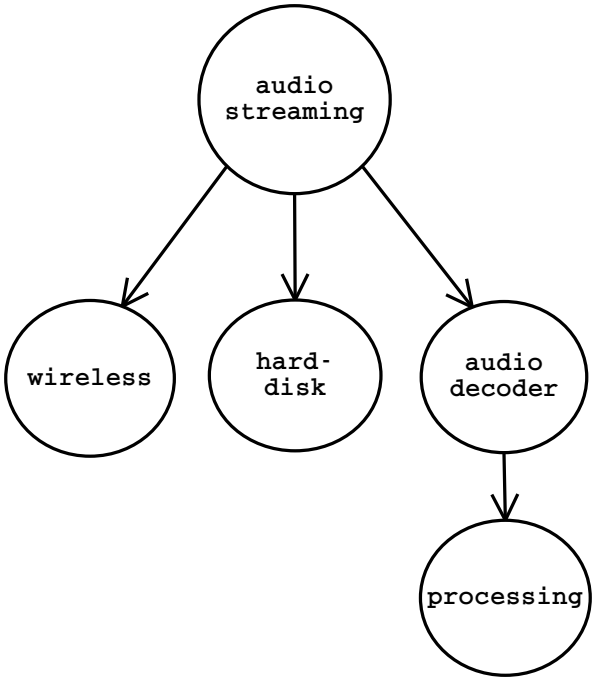


Figure 3.6. The Perfd dependency graph for the audio streaming service.

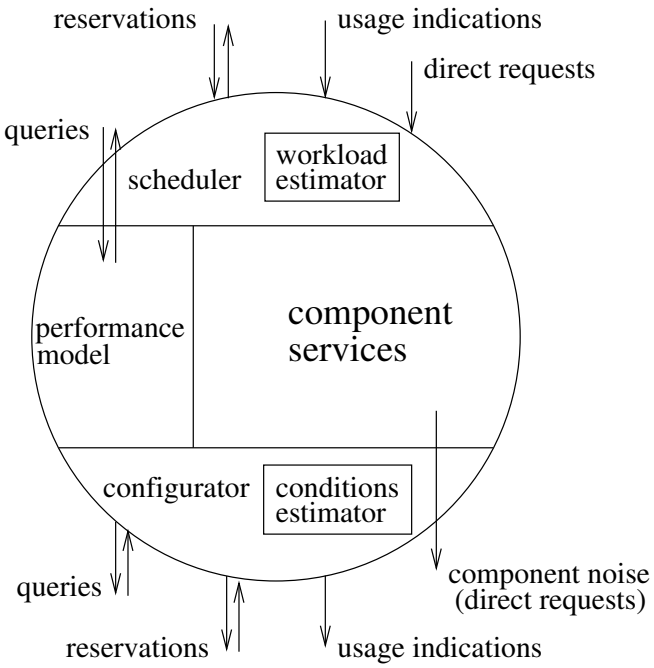


Figure 3.7. Model of a component and the information flows within Perfd.

indications (one way) are used to improve the power efficiency of the generated access schedule. The direct requests for service receive traditional best-effort service. Queries from the higher layer are directly passed on to the performance model.

The configurator exchanges information with underlying components. When the component services use other components without a prior indication or reservation, that usage is regarded as component noise. Component noise produces an unpredictable workload on other components and must therefore be minimized. Hence, all direct requests are considered component noise from a higher layer.

It is important to realize that components are not static entities; they are influenced by their environment and subject to fluctuations. The environment can be, for example, an unreliable wireless channel or the end user himself. Components must therefore be capable of handling fluctuations when they request services.

The aim of the Perfd component model is to enable cooperation. Recall from the first chapter that cooperating policies are based on access to non-functional information and context awareness. Input richness is vital (Section 2.2.3) for improving the efficiency of cooperative power-management policies. In Section 2.3.2, more details on policy input information are given. Related work that tries to exploit increased input richness is limited. Several publications only deal with the exchange of quality-related information [139] or express cost in a single dimension [197]. Within Perfd, both quality and cost can be exchanged with a multitude of parameters in the style proposed by Schilit et al. [164]. In 1993 Schilit et al. proposed a general model to exchange information based on $(parameter = value)$ pairs with $parameter$ acting as the key and $value$ holding the actual data. This resembles the older Linda tuple space model [1; 31; 32]. The advantage of the $(parameter = value)$ approach is the flexibility to exchange information.

To support cooperation using $(parameter = value)$ pairs, Perfd uses a unified interaction infrastructure (Section 1.3). This interaction infrastructure is implemented as part of the Perfd coordinator (Figure 3.1). The Perfd coordinator functions as a control plane that provides a number of facilities, such as component registration, message passing, and storage of data structures. This implies that all information flows depicted in Figure 3.7 are facilitated by the single Perfd coordinator. For complex services such as the audio streaming service, the Perfd coordinator stores data structures, including the config-space graph. The Perfd coordinator contains *no* intelligence to understand the messages passed between components or to manipulate data structures. Due to this absence of central intelligence, the Perfd approach is fundamentally different from the 'central resource broker' type of organization (Section 2.4.2).

3.2.3 Performance models

An opportunity that has not been exploited yet is using performance models as an integral part of a power-management architecture. Performance models provide the means for pro-active behavior and improved power efficiency. In Perfd, each component must have an explicit performance model capturing all trade-offs. Within the Perfd framework we use the approach that a performance model is used at execution time.

Exploiting performance models at execution time is a new approach in power management. Related work is limited to a single publication by Sinha and Chandrakasan on clock scheduling [172]. They propose to augment every function or application with a simple energy model that "separates the switching and leakage components and predicts its total energy consumption" to enable "the estimation of the energy requirement of an application that has to be executed". However, their proposal to use a performance model for power management is limited because it cannot be applied in the general

case (voltage scaling only), ignores costly memory operations (no cache or main memory), cannot handle environment changes (static FFT calculations only), and is based on a level of abstraction that is too low to be efficient (the BSIM2 MOS transistor model [167]).

A performance model within Perfd predicts for each operational mode of the component and the current environmental parameters what the quality of the offered service and the required resource usage will be. For example, an audio decoder performance model states the exact relation between input bit rate, input format, required processor/memory usage, and audio output distortion. The Perfd framework requires that each component has a performance model and that this model is separated from the component services.

The mantra "to measure is to know" is used in the literature to improve power efficiency of software. In [40] a number of tools are discussed that the application programmer can use to measure the power consumption of applications at design time. Such tools aid the software design phase with concrete measurements of performance and power consumption. By comparing the power efficiency of several implementation alternatives, an informed selection can be made. The drawback of this method is the fixation of choices at design time. At design time not all information is readily available, for instance, target platform details are unknown. It is likely that the most power-efficient solution differs for various laptop, PDA, and mobile phone models.

The Perfd approach extends the design-time measuring method. Each Perfd component includes a performance model with explicit knowledge of all supported operational modes. In numerous cases the selection between alternatives during execution time improves power efficiency, compared to the fixation of choices at design time. During execution time, far more information is available. The opportunity of Perfd to postpone vital decisions until execution time makes components more complicated, but can provide significant power savings [138]. Component developers must include such execution-time decisions in the components. During design time, the component developer must explicitly encode each decision in the Perfd driver software. In some cases the best solution is already known at design time and can be fixed directly.

Within a component, both the scheduler and the configurator use the performance model to make an informed selection between alternatives. The Perfd framework defines a uniform interface to query the performance model of all components, from hardware to software.

We have identified two problems that emerge when performance models are used. The first problem is that it is hard to determine the optimal accuracy of a performance model. The second problem is that the performance of software depends on the hardware used.

The level of accuracy is an important issue for a performance model. A performance model need not give exact answers about resource usage and resource cost. Giving an exact answer is usually counterproductive as, for example, it would require significant resources to have a full performance model of the processor inside the Perfd processor driver. A cycle-accurate performance model of a super pipelined processor including caching and memory access structure is generally very complex and computationally intensive. A simulation at the gate or transistor level to obtain the exact amount of energy required for running a certain piece of code would not yield any power savings. Thus, performance models must not aim at providing highly accurate answers, but should only give answers that will likely result in real power savings. The actual required Perfd performance model accuracy remains a difficult subject.

Fortunately, our case study suggests that very accurate models are not required for power efficiency gains. A first-order estimation (roughly 10 % error) of the performance already yields a higher power efficiency than having no performance estimation at all. Thus, by providing a first-order estimation of the trade-offs in operational modes, one can already obtain efficiency gains. More detailed performance models provide additional power savings, but suffer from diminished returns.

Another issue for the usage of performance models is their dependency on both the internal state of a component and the environment, which may change in a split second. In Perfd, every performance model has full access to information such as the workload and operational conditions in order to determine the current service quality and cost.

For instance, consider the power consumption of a video decoder. The power consumption depends on the video stream that needs to be decoded, other processes running on the same processor, the content of the cache, etc. The performance of a video decoder is dependent on the processor performance and cannot be determined for a universal processor, as a mobile Pentium 4 behaves very differently from a StrongARM 1100. The Perfd solution for this hardware dependency is to have a general performance model for a video decoder and calibrate it for a particular processor, cache and memory configuration by running a small number of performance tests when the video decoding software is installed or started. It is relatively easy to measure the required processor cycles for basic video decoding operations. This need for calibration of performance models is a wider issue. The performance models of software in general need to be calibrated for the specific platform hardware when first installed or used. Another option besides measuring at execution time is using the performance model of the processing component. Understanding this performance model requires a high level of sophistication of software; hence the calibration method may be preferred by software developers.

Another example is the wireless link. Its performance and power consumption can only be accurately calculated when all characteristics of the surroundings are known, such as movement patterns and reflection characteristics. An accurate wireless link performance model also needs to know the current value and history for parameters such as signal strength, packet error rate, allocated bit rate, and remaining free bit rate. However, a simple model can also be used to predict the wireless link cost. In [58], the energy usage for transmitting an 802.11 packet of size s is measured and modelled as: $E = a * s + b$.

To give a concrete example of a performance model, we discuss the processing component. The processing component has a performance model with information on cache and main memory size and speed, the supported frequencies for the CPU, etc. A number of possible combinations of parameters, values, and units for the performance model are depicted in Table 3.1. The information within a performance model is only understandable for a component that knows the details of the processing service. For instance, to accurately estimate the cost of decoding an MP3 audio file, the audio decoder needs to know the cost of the memory reference, average cache hits, required CPU cycles, etc.

Large-scale deployment of the Perfd framework requires the fixation of interfaces to components in general (see previous section) and performance models. Within the Perfd framework, standardized performance model interfaces need to be developed for various types of components. This is a non-trivial task as it is closely related to the problem of defining a standard *functional* interface for a component, for example, how to transmit a WLAN packet or execute a video decoder.

3.2.4 Scheduling phase

During the scheduling phase, a component delivers service to one or more components. The task of the scheduler is to coordinate the access to the component services, exploit deactivated modes, and determine the performance level (Section 2.3.3). This section provides more details and discusses an example in depth.

Figure 3.8 shows a general model of a Perfd scheduler. Components that want to use the component make either a reservation or send a usage indication to the scheduler. Direct requests for service reduce the efficiency of the scheduler because they reduce the amount of a priori information. A direct request to a hard disk that is in sleep mode implies a costly activation. If the scheduler had

Parameter	Value	Unit
main memory size	32	MByte
main memory speed	50	ns
main memory access cost	5.2	mJ/MByte
number of cache levels	1	-
cache level 1 size	4	KByte
cache level 1 speed	2	cycles
cache level 1 access cost	.48	nJ/cache line
number of processor frequencies	2	-
processor speed 1	108.6	MHz
processor speed 2	217.2	MHz
current speed	217.2	MHz
processor load	94	%

Table 3.1. Possible performance numbers published by the processing component.

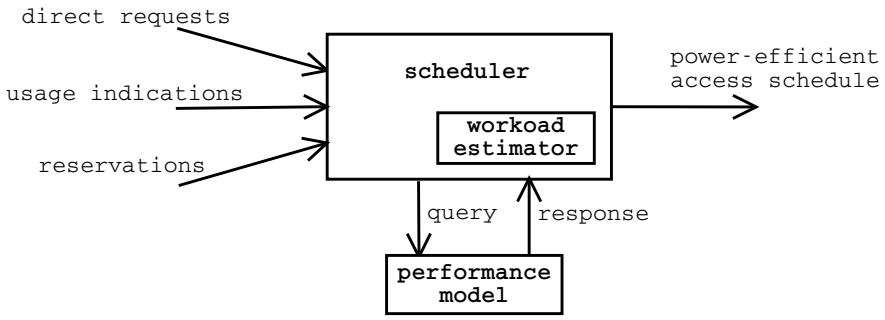


Figure 3.8. The general model of a Perfd scheduler.

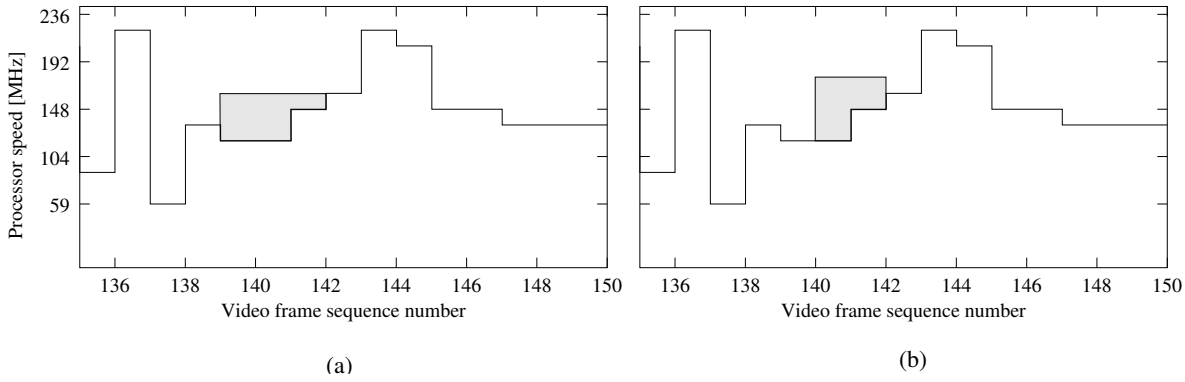


Figure 3.9. Two alternatives to schedule a new task for the Perfd processing component.

known in advance about the request through a reservation or usage indication, it would perhaps have kept the hard disk activated. Reservations and usage indications mean that the scheduler can estimate the workload in advance. The performance model is queried to determine, for instance, the power consumption in sleep mode. The scheduler adapts the operational mode of the component to the workload. Requests can be delayed in order to, for example, bundle requests together and exploit the flat utility curve of a hard disk [123].

The scheduler for a processing component will be used as an example in the remainder of this section. Chapter 5 will present detailed measurements of a Perfd processing scheduler on our LART platform. In a general-purpose OS the traditional "CPU scheduler" has the important task of ordering jobs and allocating CPU time slices both fairly and efficiently. The CPU scheduler has a significant impact on computer system performance and is the topic of numerous publications [20; 81; 141]. The task of the Perfd scheduler for the processing component is more challenging, since it must replace the traditional CPU scheduler and improve the power efficiency.

Processors have evolved rapidly from basic 8 bit processors towards super-pipelined super-scalar processors using advanced memory pre-fetching, speculative execution, and voltage scaling (Section 2.1.2). On laptops with a standard general-purpose OS (Windows, Linux, etc.) the *control* of such complex processors is becoming a bottleneck, as they can operate in many operational modes. For instance, "the PowerPC 405LP supports 6 somewhat independent parameters that affect the power consumption of the system: core voltage, CPU frequency, memory bus frequency, on-board peripheral bus frequency, external peripheral bus frequency, and LCD pixel clock frequency." [23]. Not only the control of the performance levels is a problem, deactivation modes are also more complex to control due to their fine granularity. For instance, the 405LP has the ability to deactivate individual registers.

Selecting the optimal operational mode of a processor such as the 405LP is complex. The Perfd processing scheduler is helped by increased information richness and access to a performance model. Components that require processing must hint their requirements or explicitly make a reservation. Using this performance model, the Perfd scheduler can determine the most power-efficient solution of fine-grained decisions.

For a number of tasks, such as video decoding [15; 25; 131], it can be estimated in advance what the requirements are for the processing component. The video component can send a usage indication or make a reservation for the processing component. The goal of the Perfd processing scheduler is to determine the optimal operational mode of the processor, given the processing requirements of software components. By exploiting the exponential type of utility curve of a processor (voltage scaling) it is possible to reduce power consumption significantly.

A plausible scenario is that a user is watching a movie on his/her portable device and a background task must be executed. It is non-trivial how to efficiently schedule such a new task next to the video decoding task. We implemented this scenario on our LART platform (Figure 1.8) and present actual Perfd results in Chapter 5.

Figure 3.9 shows a concrete example of the scenario with continous video decoding and a new task that must be scheduled along the decoding of video frames. The new task is short and has a certain deadline. The figure shows two alternatives for adding a new task on top of the decoding process of 15 video frames. The figure shows the minimal required processor speed to complete the video frame before the deadline given by the video decoder. Adding a new task implies increasing the processor frequency in certain intervals. The processor frequency increase is marked as a gray area in the figure. The deadline of the new task lies between the start of video frame 142 and 143. Figure 3.9.a shows the alternative where the new task is added when video frame 139, 140, and 141 must also be decoded. Figure 3.9.b shows the new task scheduled next to video frame 140 and 141. It is the responsibility of the Perfd processing scheduler to process the reservations and minimize the power consumption of the processing component. The implementation of the Perfd scheduler is non-trivial. One implementation might reason that alternative *a* is superior because the power consumption increases rapidly when the processor frequency increases. Another implementation might prefer alternative *b* because context switches and cache flushes are expensive and must be minimized.

3.2.5 Configuration phase

A coarse overview of a configurator was given in Section 3.2.2; this section provides more details and discusses the audio streaming service example. Configuration consists of two phases, as shown in Figure 3.4: an initial configuration phase and a reconfiguration phase.

The Perfd framework enables uniform treatment of reservations, usage indication, and direct requests (Figure 3.7). These three actions can be regarded as Perfd primitives. During initial configuration, only one Perfd primitive is defined, the *graph search service*, which can find the optimal operational mode for each component within a complex service.

The graph search service consists of building a config-space graph, evaluating each alternative, finding the most power-efficient alternative, and executing this alternative. Some more complex services involve multiple components, thus in that case multiple configurators must work together during the initial configuration phase.

Figure 3.3 showed the simplified config-space graph of the audio streaming service. Figure 3.10 shows a more detailed version that includes an additional trade-off between battery lifetime and audio quality. Note that for clarity only a part of the config-space graph is given, only the low-quality alternative for the quality level is expanded, all compression methods except the OGG method are not expanded, and the processing component is again omitted. A hard disk can optionally be used to cache audio tracks. Populair audio tracks are stored on the hard disk to be played multiple times without the use of the wireless link. Power savings depend on the effectiveness of the caching and cost of both wireless link and hard disk. The alternative using caching is also not expanded. In the config-space graph the different alternatives are indicated by dotted arrows, forming an “XOR” relation. Requirements are indicated with a solid arrow, forming an “AND” relation.

The quality level and compression level decision points require audio compression knowledge within the top-level audio streaming component (Figure 3.6). An audio compression method using heavy compression, such as OGG, requires a lower bit rate to obtain “FM quality” than MP1 [18; 143; 142]. Table 3.2 contains possible parameter values for the config-space graph as parameters for both the wireless link component and the decoders. Four levels of audio quality are listed in the table.

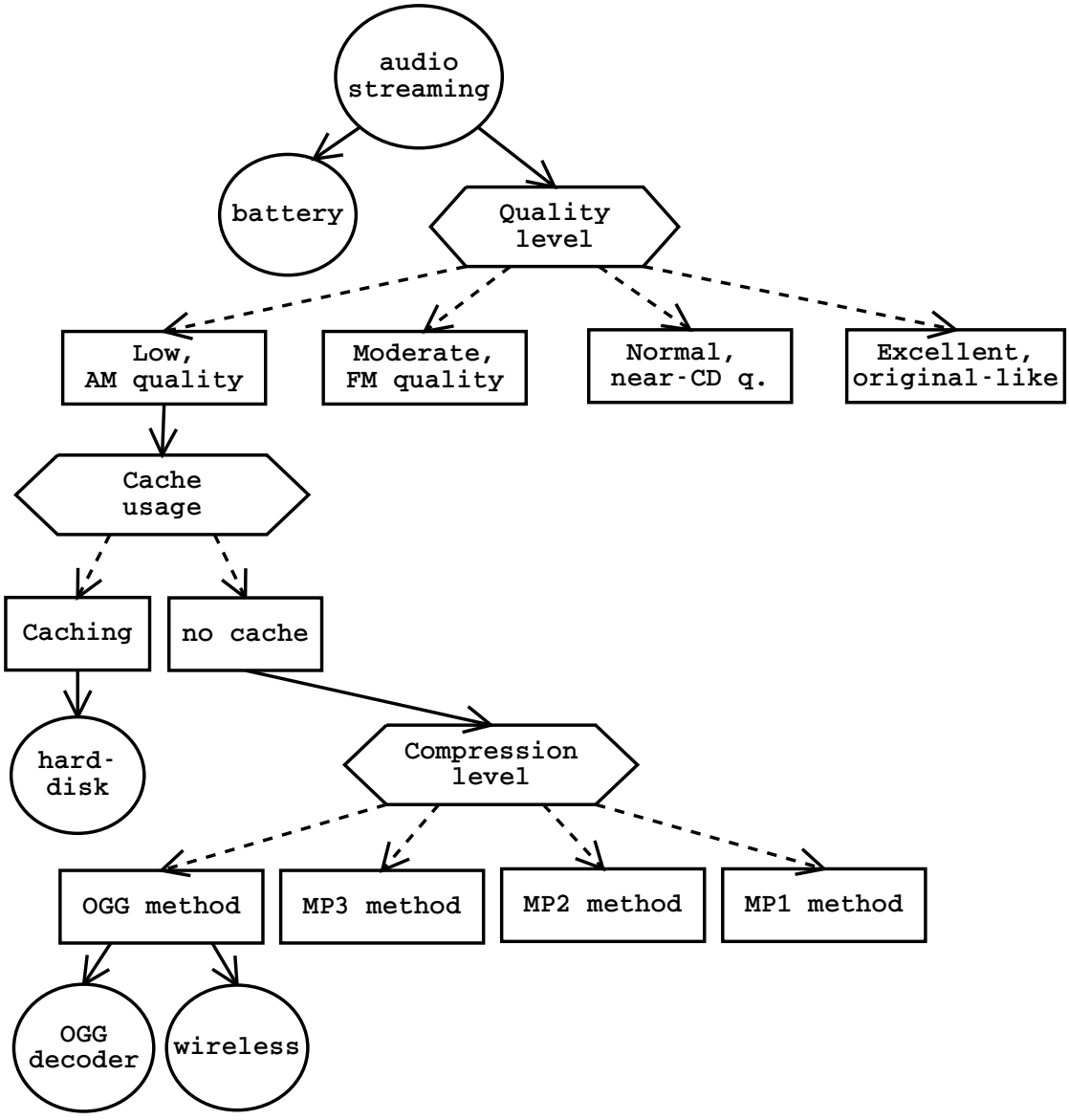


Figure 3.10. A more detailed config-space graph for audio streaming.

Compression method	Audio quality level			
	low	moderate	normal	excellent
	Kbps	Kbps	Kbps	Kbps
OGG	18	36	72	144
MP3	24	48	96	192
MP2	48	96	192	384
MP1	72	144	288	576

Table 3.2. Bit rate parameter values for the audio streaming example.

The lowest audio quality will use small amounts of both wireless bandwidth and CPU resources for decoding. The table also shows that to produce audio of the same quality, the MP3 method requires 33 % more bandwidth than the OGG method. The MP2 and MP1 method double and triple the bandwidth requirement respectively when compared to MP3.

A config-space graph is created by the cooperating configurators and stored by the Perfd coordinator. Five different actions are distinguished. The configuration process is initiated by a top-level component that requests a complex service and provides a number of parameters for that service, such as filename, minimum sampling frequency, play list, IP number of server, etc. The following five actions will be explained in detail.

1. expansion
2. evaluation
3. execution
4. monitor
5. clearing

Perfd approach

The first action in the Perfd configuration phase is the expand action: the Perfd coordinator travels downwards from the top-level component and calls each component to expand the config-space graph with both alternatives and required other components. For instance, the audio streaming component inserts the quality level decision point and the OGG decoder inserts that it requires the processing component. After the expansion action several sets of alternative *input* parameters are known for all components.

In our example the audio streaming component is the top-level component. It is responsible for translating a request such as $(length = 135min, list = /playlists/mozart\_daft\_punk.m3u)$ into parameters for the underlying components. This means understanding the trade-off between audio quality and bit rate as shown in Table 3.2.

Second is the evaluation action: the Perfd coordinator travels upwards from the bottom level in the config-space graph and requests each component to calculate the required resources for the given input parameters and current environmental conditions. The goal is to identify the most power-efficient configuration.

A *service setting* is a limited config-space graph where one alternative is selected for each decision point. Figure 3.11 shows a service setting. The audio streaming config-space graph can be converted

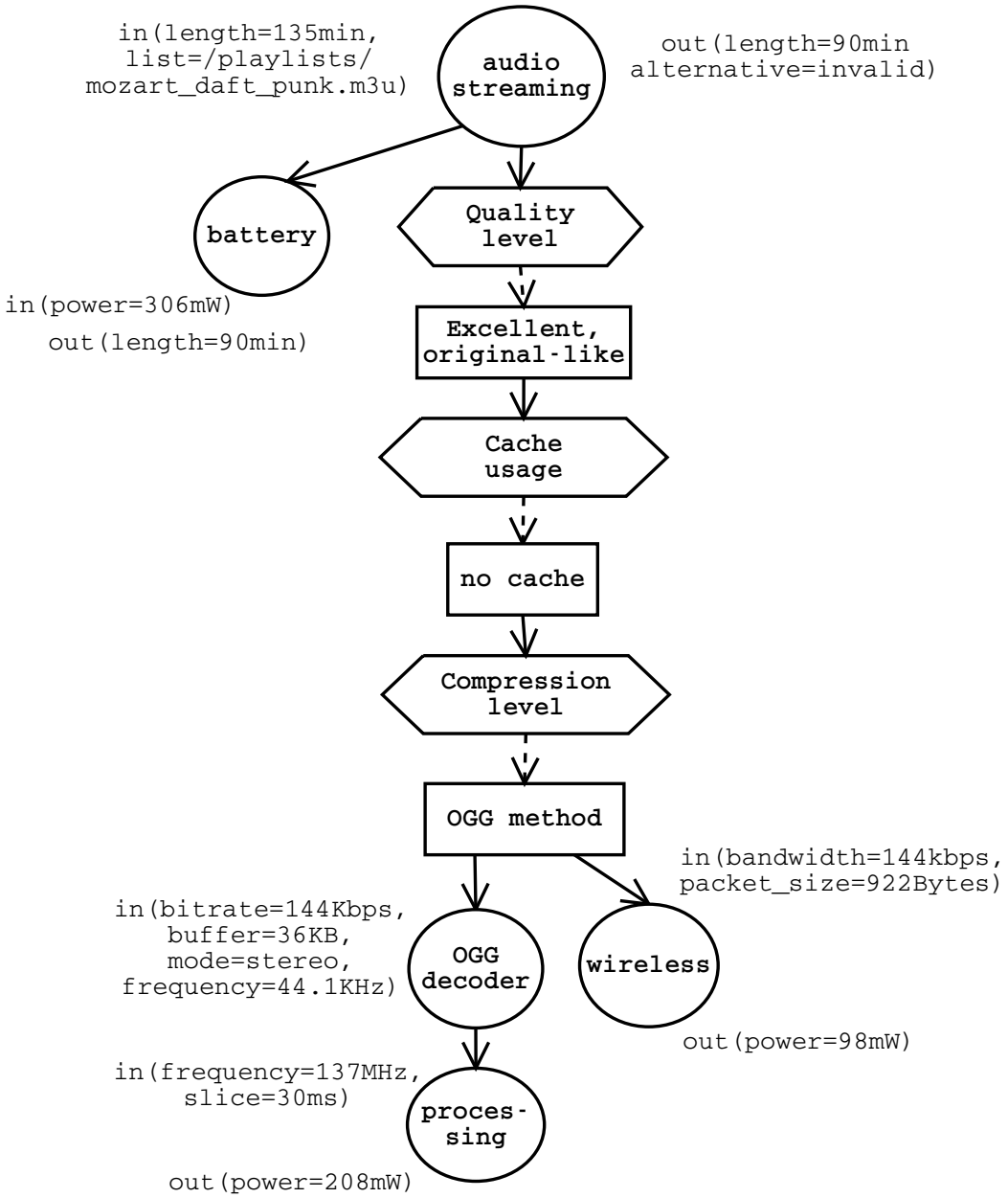


Figure 3.11. A service setting for the audio streaming example.

into several unique service settings. Each of the 32 ($4 \times 2 \times 4$) unique service settings has a different approach to offering the audio streaming service and the battery lifetime will be different for each as a result. During the evaluation action the battery lifetime for all service settings is determined and the best alternative is identified.

Figure 3.11 shows one of the 32 service settings. The most resource-intensive alternative is selected for the indicated quality level, caching is disabled, and the audio compression method with the highest compression ratio is used. Example input and output parameters of each configurator are included in this figure. For instance, the battery lifetime must be at least 135 minutes and the OGG decoder has to have a bit rate of 144kbps. The processing configurator input is the required average frequency of 137 MHz for the OGG decoder and the requirement that the decoder is called at least every 30 ms to avoid an output buffer under run. Based on this input, the configurator estimates a power consumption of 208 mW. When the Perfd coordinator travels upwards from the lower level during the evaluate action, the resource usage of "AND" components is summed. For instance, the power consumption of the wireless link (98 mW) and of the processing (208 mW) are added together and used to determine the battery lifetime.

The input parameters of the audio streaming service state that a minimal guaranteed battery lifetime of 135 minutes is required. The battery component in the service setting estimates a battery lifetime of only 90 minutes, given the calculated total power demand. Because this particular service setting does not meet the requirements it will be marked by the audio streaming configurator as invalid.

The motivation behind separating the above expand and evaluate actions instead of integrating them is the fact that the services of one component may be consumed by multiple components, resulting in multiple service access.

Multiple service access may result in *overbooking*. Overbooking occurs when within one service setting the total utilization of a component is above 100 %. For instance, a service setting that includes both an audio decoder and a video decoder may not allocate more than a total of 100 % processing capacity. Figure 3.12 shows a config-space graph where the processing component is shared amongst two other components. The multiple service access is shown as two solid arrows pointing towards a single component. The total amount of claimed processing utilization is above 100 %. The processing component configurator detects the overbooking and marks the relevant service setting as invalid.

Specific utility curves present another complication for multiple service access. For instance, due to the flat utility curve of a hard disk, multiple reads by multiple components are almost as expensive as a single read by a single component. This must be taken into account when calculating the resource usage.

The execution action, the third one, follows the expand and evaluate actions. During the execution action the best service setting is finalized by making reservations and initiating services. The valid service setting with the highest quality that meets the battery lifetime demand is selected for execution. From the top level downwards, all reservations (or usage indications) are made and services are started with the parameters stored in the config-space graph. After the service is initiated, the scheduling action is started.

The fourth action is the monitor action, in which service delivery is monitored and changes are made to a service setting when needed. The component that offers a service indicates the current quality and cost of the service. The components that consume this service are responsible for detecting changes and reacting to changes. Thus, in Perfd, a service-consuming component (client) must regularly poll for changes and respond to changes. This design decision is motivated by the fact that the majority of applications cannot adapt at arbitrary points in time. Applications such as multimedia streaming have discrete *adaptation points* at which resolution, color depth, and other parameters can

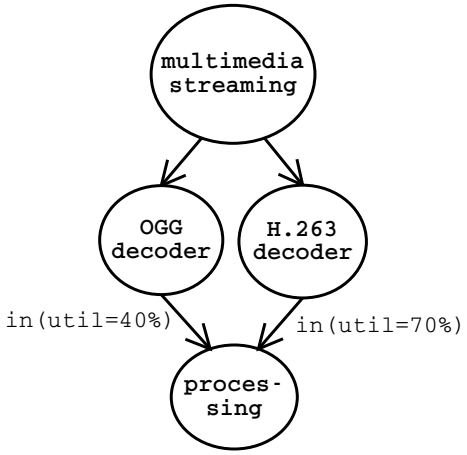


Figure 3.12. An example of two components overbooking the processing component.

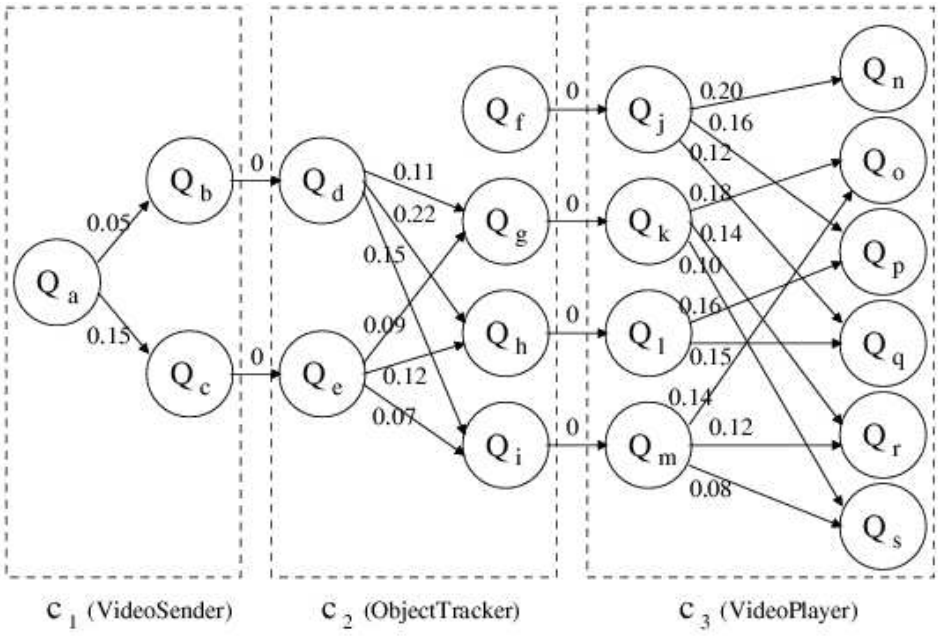
be modified. At each potential adaptation point, the configurator polls for changed quality/cost and modifies parameters when a change is required to remain power efficient. An alternative to a regular poll by the service consumer is the callback method [139]. This method uses a callback function that is executed when the component that offers a service detects changes in the quality and cost of its service. For instance, in an audio streaming scenario an audio streaming component registers a callback function called link_change() with the wireless link component. This callback function is called when the conditions on the wireless link change. Due to the existence of adaptation points in most applications, the callback method is less usefull.

The fifth and final action is the clearing action. After a service has been delivered, the involved components may directly either deactivate or reduce the performance level of the involved hardware. The traditional solution based on time-outs is less power efficient due to longer reaction times of policies. When policies know more details of the workload, time-outs are no longer required. The clearing action is relatively easy to implement compared to the other actions.

Related work

The best example of configuration-phase related work is a framework by Klara Nahrstedt et al. [193; 197], which is similar to the Perfd configuration graph-search service. In [193] Xu, Nahrstedt, and Wichadakul present an innovative multi-resource reservation framework. The framework is aimed at a distributed services context, for example, “media data distribution and processing, E-commerce, and virtual scientific laboratory services”. This framework uses a “QoS-Resource Graph” (QRG) to calculate the configuration for distributed service components.

A QRG used as an example in [193] is shown in Figure 3.13. A QRG contains several configurations of a service. Each path from Q_a to $\{Q_o, Q_p, Q_q, Q_r, Q_s\}$ represents a valid configuration for this particular example service. Within the service, resources are used of three components: the VideoSender, ObjectTracker, and VideoPlayer. Below the QRG graph itself, the compatible settings between the three components are listed. The circles in the components indicate an operational mode, such as 160 x 120 resolution or 3 trackable objects. The lines between the circles indicate which set of operational modes are valid for the three components. For each line the level of resource usage for that particular component is indicated. For instance, the line connecting Q_a with Q_c has a resource usage of 0.15 for component C_1.



Q_a = [24 frames / second , 320 x 240]
$Q_{b,d}$ = [24 frames / second , 320 x 240]
$Q_{c,e}$ = [24 frames / second , 160 x 120]
$Q_{f,j}$ = [24 frames / second , 320 x 240, 3 trackable objs]
$Q_{g,k}$ = [24 frames / second , 320 x 240, 1 trackable obj]
$Q_{h,l}$ = [24 frames / second , 160 x 120, 3 trackable objs]
$Q_{i,m}$ = [24 frames / second , 160 x 120, 1 trackable obj]
Q_n = [24 frames / second , 320 x 240, 3 trackable objs, 2.0 seconds]
Q_o = [24 frames / second , 320 x 240, 1 trackable obj, 2.0 seconds]
Q_p = [16 frames / second , 320 x 240, 3 trackable objs, 5.0 seconds]
Q_q = [24 frames / second , 160 x 120, 3 trackable objs, 2.0 seconds]
Q_r = [16 frames / second , 160 x 120, 1 trackable obj, 2.0 seconds]
Q_s = [24 frames / second , 160 x 120, 1 trackable obj, 5.0 seconds]

Figure 3.13. A QoS-Resource Graph (QRG), taken from [193]

Each path in a QRG offers a unique combination of quality and resource usage. Each path forms an alternative service configuration. The alternative configuration with the smallest maximum share of all resources is considered as "the best" configuration in their approach [193]. For instance, using 50 % of the network bandwidth and 50 % of the capacity of a proxy ($max = 50$ %) is preferred to using 10 % bandwidth, 60 % local CPU, and 10 % proxy ($max = 60$ %). A difference between the Perfd graph search service and the QRG concept is the notion of alternative configurations. Within QRG, unlike Perfd, an alternative configuration must consist of the same components. For example, in Perfd it is allowed that in one alternative configuration the wireless link component is used, instead of the hard disk.

The related work to the reconfiguration phase can be found in the area of adaptive systems [2; 113; 139], without a strong connection to power management.

The configuration phase is part of the larger Perfd framework. Related work that also uses a single framework for resource allocation, configuration, monitoring, and control comes from outside the field of power management. In [46] a single framework is presented to manage the resources in a computational grid [61], similar to the unified Perfd approach.

3.3 Implementation

We have created a wireless multimedia device with a prototype implementation of the Perfd framework. The goal of this implementation is to obtain actual power consumption numbers on real hardware under realistic workloads. The Perfd implementation allows us to compare actual power consumption numbers for certain scenarios and to evaluate its effectiveness. Our evaluation work represents only the first step towards the realization of a next-generation of power management. Every hard-disk type, processor, audio decoder, etc. needs to be modified to reap the full potential of cooperative power management.

3.3.1 Deployment

It may take several years before every piece of hardware and software inside a portable device is enhanced with support for cooperative power management. A new power-management architecture requires support for a gradual transition and must still offer certain benefits as discussed in the research challenges section of this chapter.

Figure 3.14 shows a possible gradual transition path between the current ACPI standard for power management and the cooperative power management of the Perfd framework. The left hand side of the figure shows the current situation with the 3-layer organization of ACPI. The middle of the figure shows the Perfd framework with Perfd-enhanced hardware components, but *without* the Perfd coordinator. The right-hand side shows the full Perfd deployment where monolithic applications are broken up in cooperative components.

The usability of Perfd is improved by its gradual deployment. In the first phase, hardware components are enhanced with performance models and schedulers that can exploit reservations as well as usage indications. Applications can directly communicate to such Perfd enhanced hardware components to query for the cost of hardware usage and to indicate their future workload. Every hardware component requires a modified device driver and a Perfd driver to calculate the access pattern. In this first phase it is already possible to improve power efficiency for monolithic applications such as video decoders with only minor modifications. In the second phase, the Perfd coordinator is added and the graph-search Perfd primitive can be used. Applications are broken up into components with a

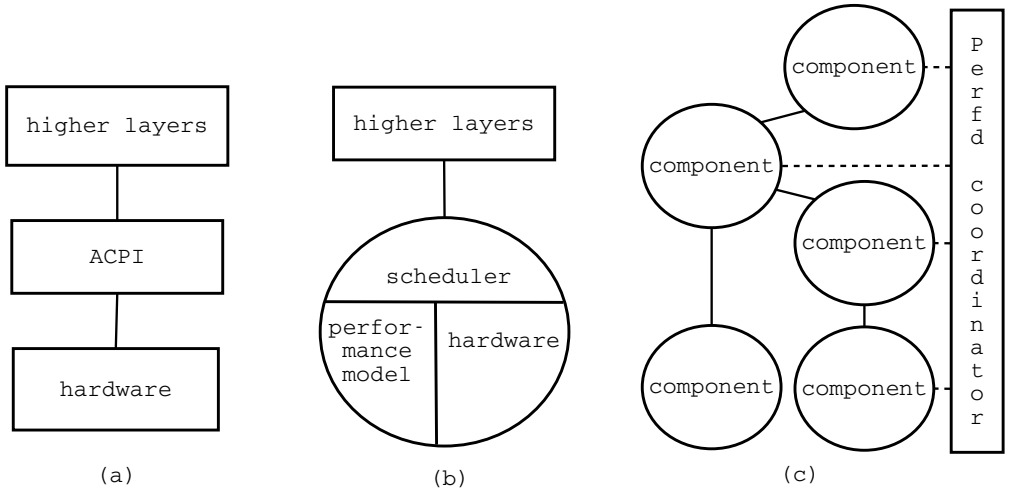


Figure 3.14. The gradual transition of power management from ACPI (a) to Perfd in two phases, (b) and (c).

configurator that has the ability to manipulate a config-space graph. With this full Perfd deployment the determined battery lifetime option is made possible (Section 1.4.2).

The next chapter describes Perfd in full deployment. Chapter 5 describes an experiment of the first phase of deployment, i.e. without a Perfd coordinator.

3.3.2 Hardware platforms

Two portable devices have been used for the experimental validation of the Perfd framework. Figure 3.15 shows them both, including an AAA sized battery for size reference. Both platforms are small processor boards capable of running from a battery. The board shown on the left, called LART, is small and optimized for low power consumption. The board shown on the right, called Assabet, is slightly larger and has numerous I/O options such as an integrated display with touch screen, an IrDA port, a USB port, and a Compact Flash interface.

The LART board forms the heart of the augmented-reality portable device that we developed within the UbiCom project at Delft University of Technology [47; 109; 183]. The LART is a custom board design by Bakker [8; 9; 11]. LART has a size of 10x7.5 cm, a weight of 50 gr, 32 MB of volatile memory, 4 MB of non-volatile memory, a SA-1100 190 MHz processor, and several I/O capabilities. All the LART design schematics and kernel modules are publically available [10]. The LART has a programmable voltage regulator for the processor voltage. The LART runs under control of the Linux operating system, which we have modified to support frequency and voltage scaling.

The Assabet platform is the Intel "microprocessor development board" for the SA-1110 processor. This board was kindly donated by Intel to Delft University of Technology. The Assabet also runs Linux, but it is not capable of voltage scaling. The SA-1110 is a slightly upgraded version of the SA-1100 processor used in the LART. The SA-1110 has a memory controller that supports more memory types.

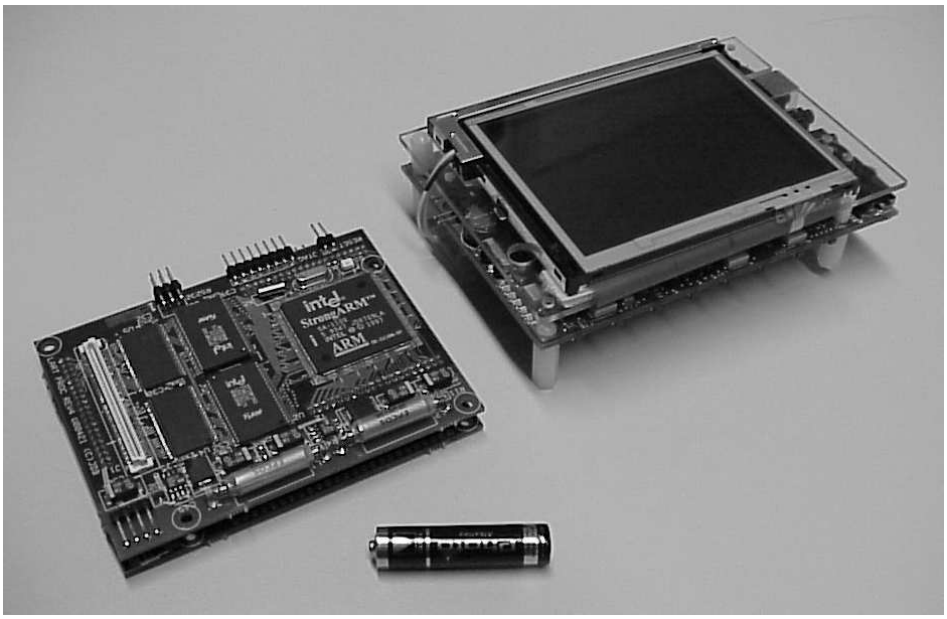


Figure 3.15. The LART and Assabet experimental platforms.

3.3.3 Power measurement

Measuring the power consumption of portable devices is difficult. In publications different methods are described to estimate the power consumption and they differ significantly in the level of accuracy. Perhaps not surprisingly, publications by software-oriented scientists tend to have a lower accuracy than publications by hardware-oriented scientists.

The power measurement equipment used in the experiments of Chapter 4 is a custom design by Bakker and Haratcherev, shown in Figure 3.16. The figure shows the measurement equipment (right) connected to a LART board (top) and some external voltage regulation hardware (bottom left). We were able to break down the total power consumption of the LART using 8 high-speed analog sampling channels (15 KHz). Additionally, the measurement setup has 3 digital channels for signalling purposes, such as indicating the start of an application or the beginning of a video frame.

To measure the power consumption of the LART (Chapter 5) we used the configuration shown in Figure 3.17. The unregulated power of a battery is converted into a fixed 3.3 V for all the components on the board, except the processor core (CPU + cache), which is supplied by a variable voltage regulator. The fixed/variable voltage and current were sampled using a small sense resistor. The standard deviation of the measurements is within 2 % of the mean.

Numerous related power-management publications use simulations [70; 186] to evaluate their ideas or use actual hardware where only the total power consumption is measured [36; 55; 119]. A complex and accurate power measurement method is presented by Chang et al. in [35]. They are able to measure the power consumption of a single CPU instruction. This level of accuracy is fortunately not needed for our purpose. In [60], the total power is measured and CPU profiling is used to break up the power consumption. The uncalibrated battery level as reported by Apple and Palm Pilot hardware is used in some publications to conduct power consumption measurements [55; 119].

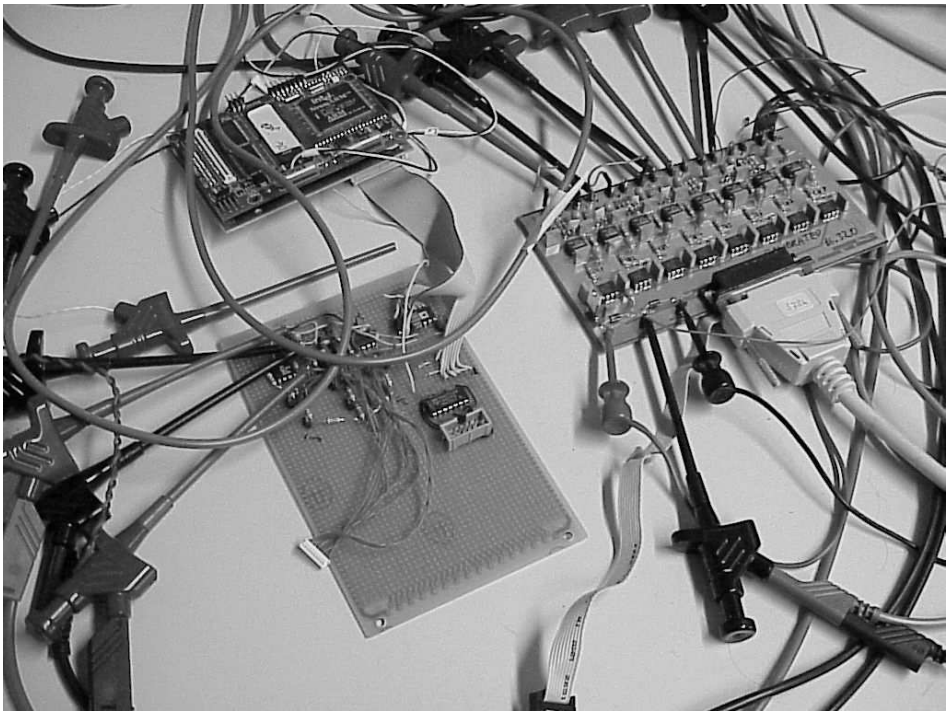


Figure 3.16. The power consumption measurement equipment.

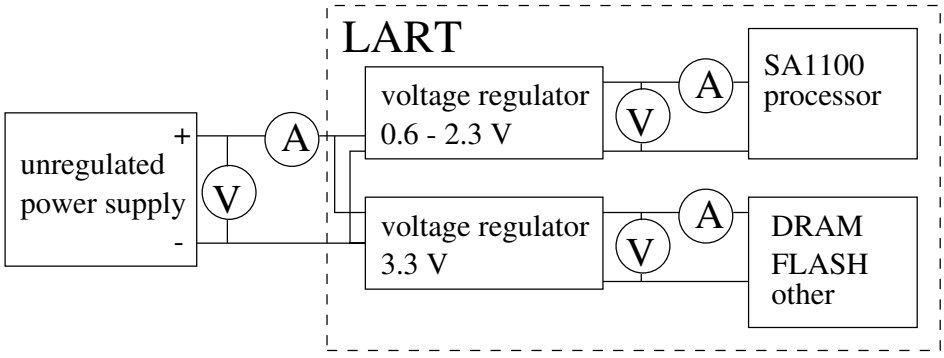


Figure 3.17. The setup for the current and voltage measurements on the LART platform.

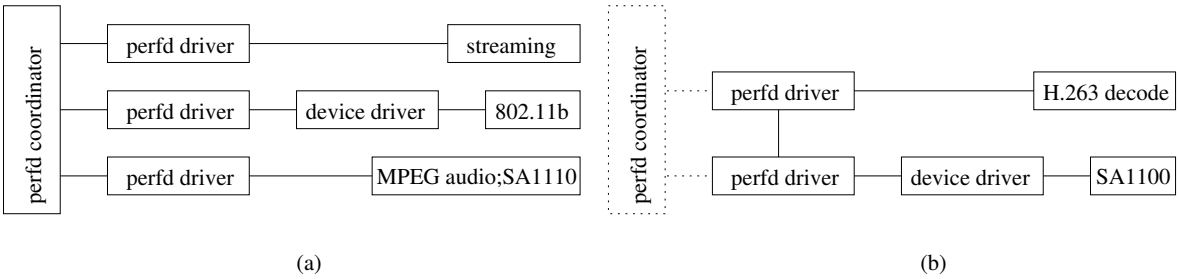


Figure 3.18. An overview of the Perfd framework implementation on the Assabet (a) and LART (b).

3.3.4 Software

The software of the Perfd framework consists of the Perfd coordinator, Perfd drivers, modified OS drivers, and software components. Figure 3.18 shows an overview of the components in the implementation. The left side shows the audio streaming service implementation on the Assabet platform using a wireless link, streaming, and MPEG audio decoding on a SA-1110 (Chapter 4). The audio decoding and processing are joined together in a single component for simplicity. Chapter 5 describes the processing component in detail. The right side of Figure 3.18 shows the processing scheduler experiments on the LART platform. Both the H.263 video decoder and the SA-1100 processing component are connected to the Perfd coordinator using a dotted line. Our experiments with the video decoder and the SA-1100 processor were done in the spirit of the first phase of the Perfd deployment plan, where modified applications directly communicate with enhanced hardware components.

The Perfd coordinator is implemented in C code as a user space program that can be extended during execution time. The Perfd coordinator uses dynamic library loading to incoperate new Perfd drivers at execution time. The interface to the Perfd coordinator is implemented using UNIX sockets. The Perfd coordinator can be used manually by a command line client. Figure 3.19 shows a screen shot of a Linux terminal where the Perfd coordinator is started as a background process and a command line client runs in the foreground. The Perfd coordinator supports commands to load and unload Perfd drivers, query performance models, reserve resources, indicate usage, and initiate a configuration action.

Drivers

Figure 3.18.a shows the wireless Perfd component. This Perfd driver uses an 802.11b Compact Flash WLAN card. The performance model answers queries on the power consumption for actions such as a single packet transfer, streaming of packets, and entering sleep mode. Note that the energy per MByte is dependent on the signal strength as 802.11b uses automatic rate selection (Section 2.1.3).

The streaming component controls the 802.11 and decoding component. The decoding component uses the Madplay MPEG audio decoder to decode the compressed bit streams. Madplay is a standard Open Source MPEG audio decoder that uses only fixed-point calculations to decode MP1, MP2, and MP3 bit streams. Floating-point operations are not supported in the SA-1100 processor instruction set and a fixed-point decoder is therefore a necessity for performance.

The processing component in Figure 3.18.b is built around the SA-1100 processor. We created a Linux OS device driver that changes the clock frequency, changes the voltage, and subsequently recalibrates the kernel's internal delay routines, in particular those that busy-wait by counting instruction

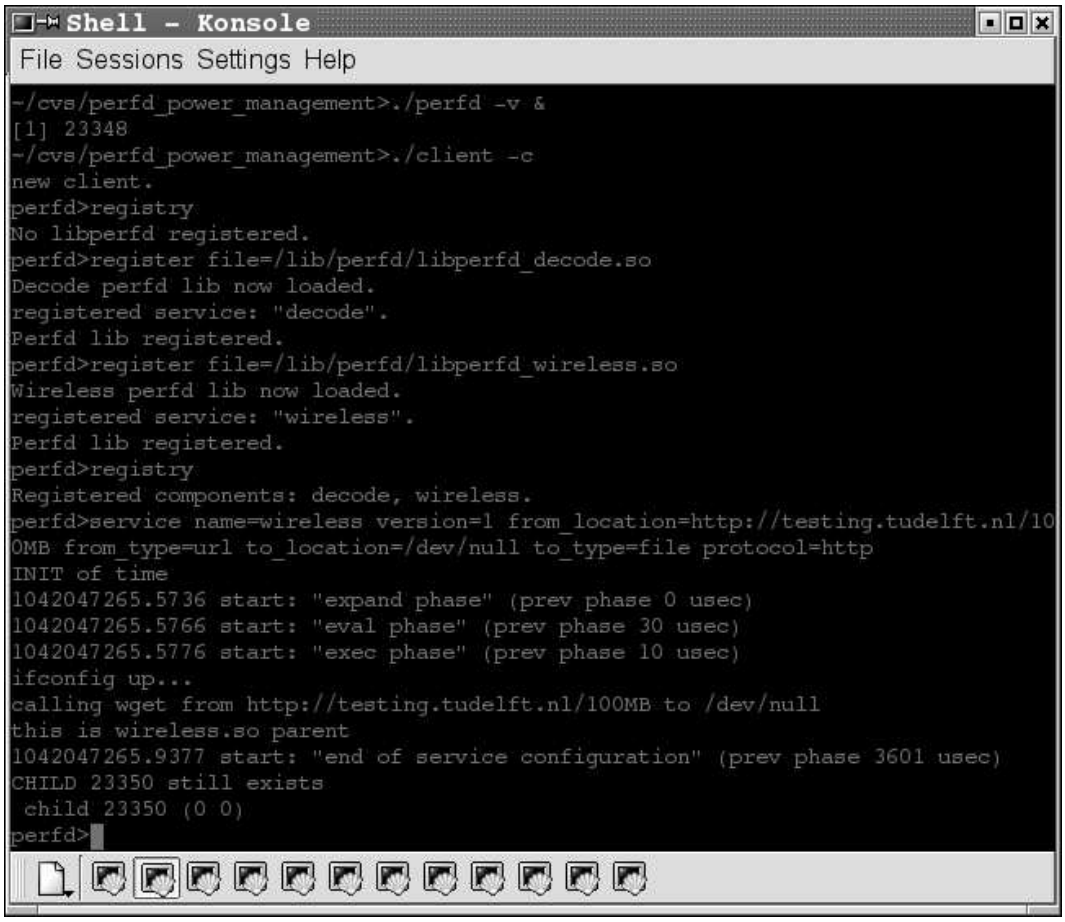


Figure 3.19. A screen shot of the Perfd coordinator usage.

cycles. In addition, the device driver adjusts the memory parameters that control the read/write cycles on the external bus. The code has been structured such that it may be interrupted and does not depend on off-chip memory, which is temporarily unavailable during a clock frequency change.

This frequency/voltage scaling device driver was written in 1999 by the author. At that time the LART was the first research platform capable of voltage scaling while running a general-purpose OS. The SA-1100 does not officially support voltage scaling (Section 2.1.2). The Crusoe processor was the first commercial processor to explicitly support voltage scaling (available in 2000).

The initial driver from 1999 was generalized by using a pre-change and post-change event notification system. With such a general event notification system it is possible to pause and resume actions that are sensitive to frequency changes, such as DMA transfers. Our initial driver helped seed the "CPUFreq" project started in 2001 that consists of several volunteers that aim to provide Linux support for all voltage-scaling-capable processors. The project's patches to support frequency and voltage scaling for the StrongARM, PowerPC, SuperH, and x86 processors were accepted by Linus Torvalds in 2002 for the 2.5.x developers kernel.

Perfd coordinator

The Perfd coordinator offers several support services for the Perfd drivers and applications. This section will explain some of these features down to the most basic level, using C code for illustration.

First, and foremost, the Perfd coordinator offers a unified interaction infrastructure based on message passing. Second, it stores the data structures that the Perfd drivers use for configuration and scheduling. Third, it parses service requests, reservations, usage indication and performance model queries, allocates special data structures for their storage, and invokes the relevant Perfd driver function. Thus, Perfd drivers are activated exclusively by the Perfd coordinator, which handles both communication and storage.

The Perfd coordinator is implemented using the UNIX daemon paradigm. The Perfd coordinator daemon is contacted using a predefined fixed socket number. A client such as a Perfd enhanced application binds to this socket and sends commands. During system initialization, all Perfd drivers must be registered with the Perfd coordinator by using startup scripts. Several applications can be connected to the Perfd coordinator socket. Their requests are handled in the order of their arrival to avoid overbooking and to minimise the complexity. Priority levels or other solutions can be used by Perfd drivers to single out critical tasks. The priorities for configuration or scheduling can be added to the $parameter = value$ interface. Due to the modular nature of the Perfd framework it is possible to add a priority system to a new interface version of a component without affecting the Perfd coordinator. It is even possible to load a Perfd driver with priorities without unloading an older version with less sophistication. This feature hopefully ensures that Perfd drivers are easier to upgrade than, for example, DLL files on the Windows OS.

A Perfd driver is registered by sending the *register* command to the Perfd coordinator together with a single $parameter = value$ combination to pass the filename of the Perfd driver. Perfd drivers are implemented as dynamically loaded link libraries. After the Perfd coordinator reads a command such as *register file=/lib/perfd/wireless.so* from the socket, this library is loaded using the `dlopen()` call to the Linux dynamic linking loader. The Perfd drivers are imported in the same address space as the Perfd coordinator.

Every Perfd driver has a predefined registration function. After the wireless.so file is opened, the registration function `libperfd_register` is called. This registration function has the task of adding its component name, component version, and interface version to the `perfdcomponent_t` structure. This structure also stores handlers to the important Perfd driver functions. Filling in these

handlers is the responsibility of the Perfd coordinator. The `perfdcomponent_t` structure is defined as follows:

```
struct perfdcomponent_t;

typedef void (*perfdregister_t)(struct perfdcomponent_t*);

struct perfdcomponent_t {
  char                *service_name;
  int                 component_version;
  int                 interface_version;
  void                *component_handle;

  perfdregister_t     register_handle;
  perfdexpand_t       expand_handle;
  perfdevaluate_t     eval_handle;
  perfdexecute_t      exec_handle;
  perfdperfmodel_t    perfmodel_handle;
  perfdreserve_t      reserve_handle;
  perfdindicate_t     indicate_handle;

  struct perfdcomponent_t   *next;
};
```

In order to be as generic as possible, the Perfd coordinator provides a maximum amount of freedom for implementing a distributed power management policy. The Perfd framework only defines how and when the seven handle functions in the Perfd driver will be called.

The wireless.so Perfd driver defines the mandatory registration function using the following lines of code.

```
const char *SERVICENAME = "wireless";

void libperfd_register(struct perfdcomponent_t *admin) {
  admin->service_name = (char *)SERVICENAME;
  admin->component_version = 1;
  admin->interface_version = 1;
}
```

The unique combination of service_name and component_version is used with the Perfd coordinator to address the different Perfd drivers, both are therefore mandatory. When invoking services, placing reservations, or indicating usage, the argument *name=wireless version=1* is used to address our example Perfd driver.

The services of components, shown in Figure 3.7, are called using the *service* command of the Perfd coordinator. To use the wireless component to transfer a test file from a web server we can use the following command *service name=wireless version=1 from_location=http://testing.tudelft.nl/100MB from_type=url to_location=/dev/null to_type=file protocol=http*. Interface version 1 of the wireless component defines the "from" and the "to" location for transport and the type of this location, for

instance, url, file, file_descriptor, or socket. This interface is easy to extend with more features such as UDP protocol support.

The Perfd coordinator parses the raw service command text and transforms it into a structure that is the basis of a config-space graph. The `perfdservice_t` structure stores all details of a service, including all alternatives that must be resolved during the initial configuration, parameters, and Linux process IDs of forked children.

```
struct perfdservice_t {
  char                        *name;
  int                          version;
  int                          power;
  int                          overbooking;
  int                          verbose;
  struct perfdcomponent_t     *component_list;
  struct perfdalternative_t   *alternative_list;
  struct perfdalternative_t   *best_alternative;
  struct perfdrequirement_t   *requirement_list;
  struct perfdparameter_t     *parameter_list;
  struct perfdchild_t         *child_list;
};
```

During the initial configuration phase, the expand action ensures that all underlying required services are added to the config-space graph. Similar to the `libperfd_register` function, every Perfd driver includes a `libperfd_expand` function. This function is called by the Perfd coordinator. The `libperfd_expand` function in a Perfd driver must add requirements and alternatives, such as shown in Table 3.2 to the `perfdservice_t` struct.

The `libperfd_eval` functions are called for all `perfdservice_t` structs in the config-space graph during the evaluation action. The power consumption values are calculated by these eval functions using their own performance model and stored in the `power` field of the struct. The alternative that is considered "best" by the eval function is stored in the `best_alternative` field. From the lowest level up to the top level the power numbers are calculated by using `best_alternative` power consumption. For example, when the config-space graph includes alternative quality levels for audio playback, the highest quality that meets a certain condition (e.g. battery lifetime) is used for both the `best_alternative` pointer and the calculation of the `power` value.

The `libperfd_execute` function of the top-level component starts the delivery of the service. In turn, the top-level component must call the execute function of lower-level components in the config-space graph. The Perfd coordinator only calls the top-level execute function because the others may operate at different time scales and frequencies. For example, the top-level streaming component calls the execute function of the decode component several times; once for every track in the playlist.

The tasks of the Perfd coordinator during the scheduling/reconfiguration phase (Figure 3.4) are less complex than those during the initial configuration phase. Most of the complexity is inside the execute function of components. The `libperfd_execute` function of the top-level component in the config-space graph implements the execute, monitor, and clear actions. The starting time of the scheduling phase is defined as: the time at which the `libperfd_execute` function causes a service to be delivered. It does not return until these actions are completed. Note that the Perfd coordinator is unaware of the exact start of service delivery.

Reservations, usage indications, and performance model queries can be issued from both the command line and from inside a Perfd driver. The *reserve* and *indicate* commands are used for reservations

Command	Description
registry	display the current registered Perfd drivers
register	register a new Perfd driver
unregister	remove a Perfd driver from the registry
service	call a service of a component
reserve	issue a reservation for a service
indicate	inform a Perfd driver of future usage
query	obtain information from a performance model

Table 3.3. Summary of the Perfd coordinator commands.

and usage indications from the command line. The *query* command is used to extract information from the performance model of a component. These commands trigger a call to the `reserve_handle`, `indicate_handle`, and `perfmodel_handle`, defined in the `perfdcomponent_t` struct. The commands must be followed by the service_name and component_version arguments and details of either the reservations, the usage indication, or the query. Perfd drivers issue a direct call to these handle functions.

All Perfd coordinator commands are shown in Table 3.3.

3.4 Conclusions

The Perfd framework is designed to partly address the challenges ahead of us in the field of portable devices. In our view the challenges are to increase context awareness, enable modularization, improve information access, and move from reactive solutions towards pro-active solutions.

The Perfd framework is designed to stimulate cooperative power management, provide a uniform framework for both resource reservations and usage indications, enable pre-determined battery lifetime, and expose the different operational modes of both hardware and software to the upper layers.

Perfd encompasses several novel research ideas. Traditional power-management solutions use a single monolitic power-management policy. The distributed power-management policies within Perfd cooperate to find the most power-efficient operational mode (Figure 3.2). The configurator/scheduler cooperation enhances both context awareness and information access. This enables power efficiency gains. Another idea included in Perfd is the use of performance models at execution time as a means to make predictions and enable pro-active behavior. Perfd also makes alternatives explict and postpones some decisions that have a high impact on power consumption to execution time, when more information is available.

We have described some of the details of our Perfd framework implementation on our embedded Linux platforms. In the next two chapters we will show how our Perfd implementation behaves on actual hardware.

Chapter 4

Configuration

THIS chapter describes a configuration experiment. The goal is to demonstrate the inner workings of the configuration phase within the Perfd framework.

We use audio streaming as a leading example for evaluation of our Perfd framework for making trade-offs. The new power management policies are context-aware, understand their environment, and cooperate to find an operational mode that minimizes the total power consumption of all the components of a portable device. A config-space graph is used to make alternative configurations explicit. The results presented in this chapter show that the Perfd framework is able to predict and select the operational mode with the lowest power consumption.

Roadmap

4.1 Scenario

This chapter focuses on a *wireless audio access* scenario, where a server on the Internet streams compressed audio information to a mobile user. The user is carrying a portable device with one or more wireless links to receive and play this compressed audio information. In our configuration experiment we use the Perfd audio streaming service to implement this scenario (Section 3.2.5).

To obtain efficient power management, one must understand the trade-offs involved in an audio streaming service so that one can select the best configuration. It might even be necessary to adapt the configuration of the service dynamically, due to the fact that the quality of the wireless link depends on the environment. The following three actions account for most of the power consumption of a wireless audio streaming service:

1. Receiving compressed audio over a wireless link
2. Decoding the compressed audio
3. Outputting the analog audio signal

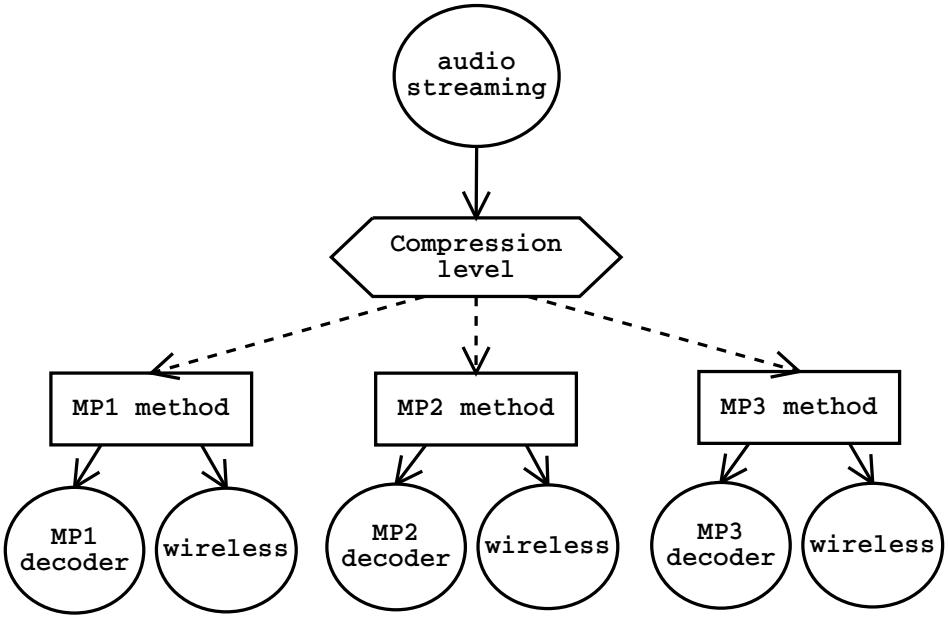


Figure 4.1. The config-space graph used for experimental validation of the Perfd framework implementation.

Within each of these actions there are trade-offs to be made on the quality and the cost of the resources involved. More interesting, however, is that trade-offs *between* the first and the second action can be made; heavily compressed audio requires extensive decoding with moderate data rates across the wireless link, while lightly compressed audio requires little processing for decoding but induces a high data rate. Which alternative is best depends on the power consumption of the available audio decoders and the effectiveness of the wireless link (at the current time) in handling the associated data rates.

This chapter describes the trade-off in power consumption between the wireless link and audio decoding. We present detailed performance measurements of an IEEE 802.11 compliant wireless Compact Flash card and an MPEG audio decoder (layers 1, 2, and 3). These performance measurements are used to create simple performance models. These models are a necessary prerequisite to determine which operational mode yields the highest power efficiency.

Figure 4.1 shows the trade-off under investigation in the form of a config-space graph. The service consists of three Perfd drivers, namely streaming, wireless, and decode. At execution time the performance models of the wireless and decode components are queried and used to determine the most power-efficient compression level.

An experiment similar to our case study is conducted by researchers from MIT [14]. This detailed study also investigates the trade-off between compression and transmission. This study is focused on the general case of data compression. The motivation of this work is that ”the energy required for transmission of a single bit has been measured to be over 1000 times greater than a single 32-bit computation”. Figure 4.2 lists some of the results from their case study. The figure shows the required energy to first receive and consequentially decompress a data file for several compression methods (bzip2, compress, LZO, etc.). The energy that is used when the data file is not compressed before transmission is also shown. The energy usage is broken down into four sources, peripherals, network,

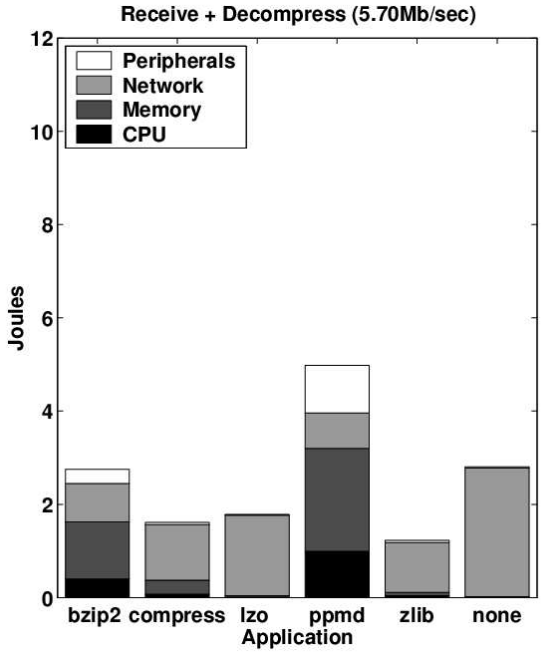


Figure 4.2. Energy for receiving and decompressing a 1 MB data file, taken from [14]

memory, and CPU. When no compression is used, all energy is spent on the "Network" and the data file must be completely transferred (1 MByte). It can be seen that heavy compression algorithms such as bzip2 and ppmd do not yield any savings, as the network portion no longer dominates the total energy usage. The energy spent on the CPU and memory nullifies all compression gains; ppmd is even highly counterproductive. Figure 4.2 shows that the zlib algorithm with its modest compression ratio and minimal processing demands is the optimal algorithm.

This case study supports our claim that awareness of trade-offs between components in a portable device can yield power-efficiency gains. This study also illustrates that exploiting trade-offs is non-trivial, even for a very specific case. This case study does not include a framework to handle trade-offs in general nor a platform-independent solution to specifically exploit compression trade-offs.

4.2 Wireless link

Figure 4.1 showed the config-space graph of an audio streaming service that includes a wireless link. This section presents detailed performance measurements and a simple performance model for a specific wireless link.

The IEEE 802.11b Wireless LAN (WLAN) standard defines a method to transfer several Mbps across a distance of up to a few hundred meters in the 2.4 GHz band (Section 2.1.3). We have used an 802.11b Compact Flash card in combination with an Assabet board to set up a wireless link in an audio streaming service. The card is a "SocketCom low-power WLAN card", using a Symbol chipset.

All WLAN measurements were taken in a situation where the sender-receiver distance was such that no re-transmissions were needed and the automatic-rate selection algorithm was disabled by fixing the data rate to 11Mbps, unless stated otherwise. The WLAN card is used in infrastructured mode and is connected to a Linksys WAP11 base station. This base station has been used because it can be re-configured using Linux-compatible software tools. Measurements were taken between the WLAN

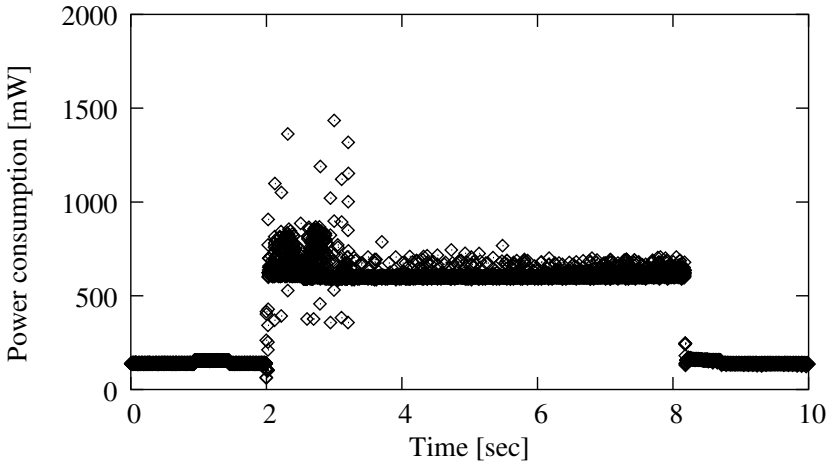


Figure 4.3. The WLAN power consumption when activated, yet without data traffic.

card and the Assabet board using a Cycard Compact Flash extender. We added a sense resistor to measure both the current and voltage of the card (Section 3.3.3).

Figure 4.3 shows the power consumption of the WLAN card when activated for a short period of time. The figure shows the results of a 10 second measurement period where the card was turned on for a few seconds and turned off again using the Linux ifconfig command. No data traffic occured during this measurement. The measurement was taken using a low sampling frequency (1.5 KHz). A high sampling rate would result in too many data points and reduce the readability of the figure. On the other hand, using averaging to reduce the number of samples would smooth out the results. The power consumption of the card when activated, but with no data traffic, was measured to be between 612 and 615 mW with a longer measurement period and a higher sampling rate (15 KHz). The power consumption when the card is deactivated is 144 mW. These numbers show that an active wireless link is very costly in terms of power consumption and that it must be turned off whenever possible.

Figure 4.4 shows the power consumption of the WLAN card when receiving data from a web server. The interval from 2.2 s to 4.2 s showing small variations in power consumption is clearly different with respect to the interval 4.2 s to 8.6 s, where the power consumption varies rapidly between 600 mW and 1500 mW. The difference between the two intervals is caused by the device driver seeking on all frequencies for the strongest base station signal and automatically connecting to that base station. The device driver also sets the network ID (ESSID) to the ID advertised by the base station. This process takes roughly 2 seconds with a few ms of variation. After that the WLAN card is receiving data at full rate. The TCP/IP throughput is around 5 Mbps and the average power consumption is 838 mW. The rapid variation in the power consumption is due to the increase in power consumption when receiving a packet.

Figure 4.4 shows a significant variation in power consumption when receiving data. In Figure 4.5 we zoom in to the ms scale. This figure shows the actual transmission and reception of two packets resulting from a network ping command. At this scale we can see that the power consumption slowly rises and falls; the sampling rate is sufficient to accurately capture events at the packet level. The two peaks are the transmission of a 256 byte icmp echo packet and the reception of the corresponding icmp echo reply packet. The 256 byte payload of the echo packet is absent in the reply packet. This difference in packet size explains the difference in size of the two peaks. Note that when no packet

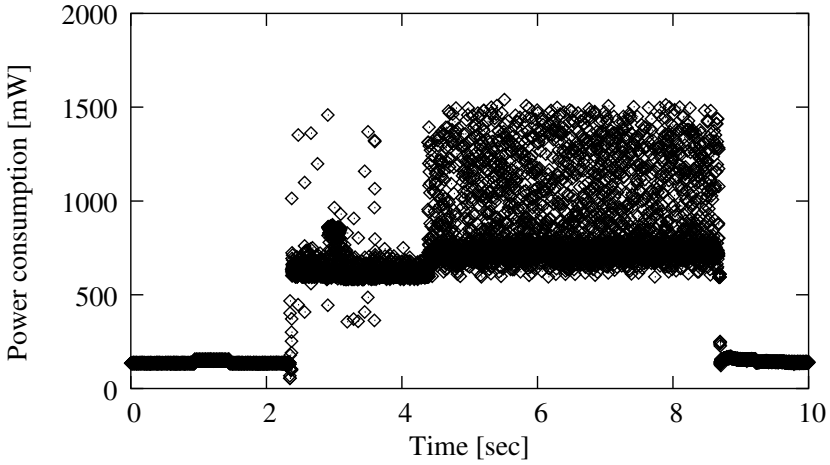


Figure 4.4. The WLAN power consumption when receiving data for several seconds.

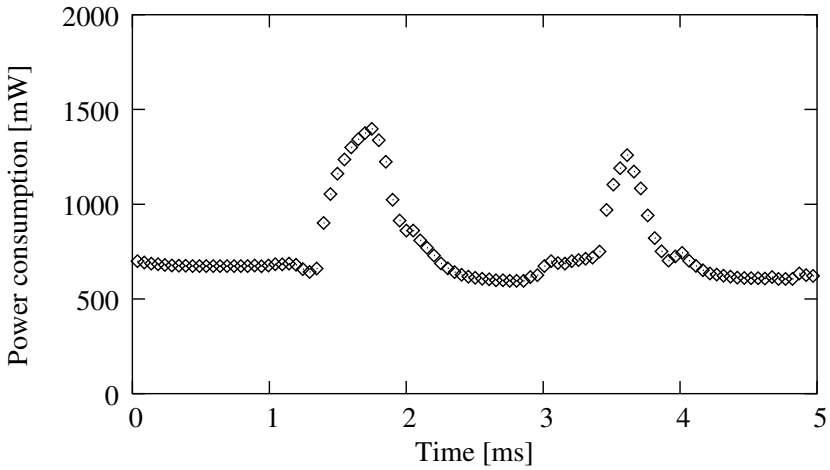


Figure 4.5. The WLAN power consumption during a network “ping” command.

is transmitted the power consumption is close to the 612-615 mW measured as idle power. The peak power consumption in this figure is close to 1500 mW, identical to the results in Figure 4.4. This explains the difference between Figure 4.3 and Figure 4.4. The individual samples in Figure 4.4 can be identified as occasional peaks, but frequently occurring values appear as a thick black band, such as the idle period from 0 to 2.2 s. The highest intensity in interval 4.2 to 8.4 lies between 600 and 700 mW; this corresponds to zero traffic as validated by Figure 4.5. The lighter area in this interval from 700 to 1500 mW is caused by the reception of data packets from the test file, the transmission of TCP acknowledgements, and HTTP control packets.

Figures 4.3, 4.4, and 4.5 all give the power consumption numbers for only the WLAN card. However, such numbers do not accurately represent the resource usage for the wireless link as a whole. Active hardware components on the Assabet when using a WLAN card must also be considered. The overhead of the software running on the Assabet when actively transmitting data is small, and can be ignored. Significant overhead is caused by the voltage conversion. The 3.3V needed for the card is converted from the variable battery voltage at an efficiency of around 80 %. The total power consumption is therefore considerably larger.

When idle the Assabet board and WLAN card consume a total of *1030 mW*. This unusually high amount is most likely due to the fact that the Assabet board is an older development board. It is expected that the new Intel XScale development board offers a lower power consumption when in idle mode. The LART board (Section 2.1.2) has an idle power consumption of only 52 mW, but currently offers no support for a WLAN card. All power consumption numbers in this chapter are heavily influenced by the high idle power consumption of the Assabet in combination with a WLAN card.

One approach to quantify the total cost of the wireless link is to precisely record for the download of a test file the start time, completion time, and the average power during this period. The energy required for the download is simply the energy spent between the start and stop time. The problem with this approach is that after the downloading has stopped and the WLAN card is deactivated, it takes some time before the Assabet system returns to the idle power consumption level of 1030 mW. We used another method to determine the cost of the wireless link. By using a given time interval for a test download, we can compare the result with the idle power consumption and use this to calculate the energy per download.

In our experiment we downloaded various test files and recorded the power consumption with a frequency of 15 KHz during 20 seconds. The test files sizes range from 0.01 MByte to 4 MByte. Perfd was used to download the test files from a web server. During the 20 second interval, the Assabet and WLAN are either in idle mode or downloading the test file. The transfer itself takes about 2 s for 1 MByte and almost 7 s for 4 MByte. The idle times are 16 s and 11 s respectively, due to the 2 s activation time of the WLAN card. The complete command as issued by the command line Perfd client (Section 3.3.4) for the 0.01MByte test file download is *service name=wireless from_location= http://130.161.43.247/0.01MB from_type=url to_location=/dev/null to_type=file protocol=http*. The Perfd driver for the wireless service exploits the deactivated mode of the WLAN card. When the wireless link service is called, the card is activated and after usage it is directly deactivated.

Figure 4.6 shows the energy usage for downloading the test files from a web server. The figure shows the energy usage for five test downloads in 11 Mbps mode and two for 2 Mbps mode.

The adaptive-rate algorithm in the 802.11 standard reduces the data rate when the standard mode of 11 Mbps yields too many packet retransmissions. The lower data rates in the standard modes of 5.5, 2, and 1 Mbps are more robust to errors (Section 2.1.3). This robustness comes at the cost of power efficiency. Transferring 2 MBytes when in 2 Mbps mode uses 12400 mJ, which is 7800 mJ more than in 11 Mbps mode. In earlier research we investigated the effect of receiver-transmitter distance on the link quality [150]. A full analysis of WLAN performance is outside the scope of this chapter.

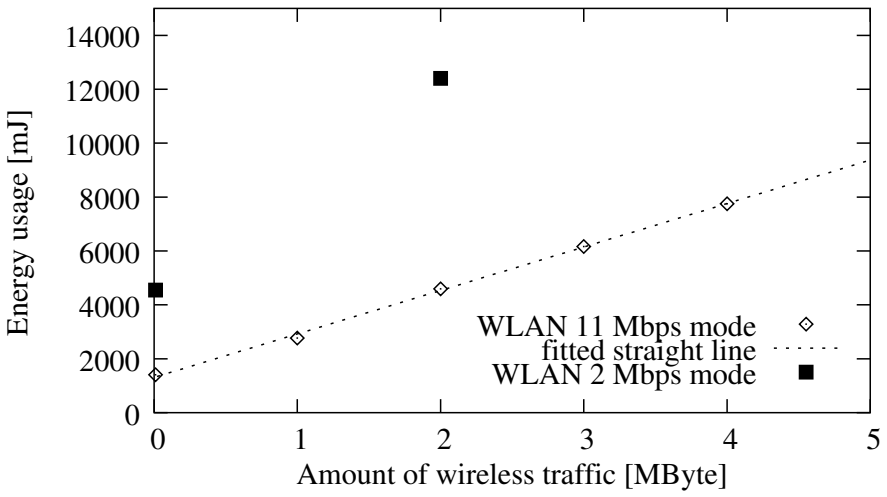


Figure 4.6. The performance model for downloading data using Perfd.

Also shown in Figure 4.6 is a straight line that best fits the measurements in 11 Mbps mode; the variance of residuals is a mere 0.011 (reduced chi square). The figure shows that a very simple performance model can already accurately predict the wireless link performance. From this simple line fit we obtain the following performance model that states the relation between energy usage and download file size (S_{file}, in MByte).

$$E_{802.11\ 11Mbps} = 1310 + 1610 \cdot S_{file} \tag{4.1}$$

4.3 Audio decoding

The config-space graph of Figure 4.1 shows three different audio decoders. This section analyses the power consumption of these decoders on our Assabet platform.

The power consumption of the Assabet during audio decoding and playback is composed of the cost of decoding and the cost of analog audio output. Figure 4.7 shows that the analog audio output cost cannot be neglected. The figure depicts the power consumption of the Assabet when generating sound for about five seconds. From 0 to 2.5 s the Assabet is idle. After that an audio test program is executed. This program generates a buffer with a test signal and copies this buffer repeatedly to the Linux audio output interface /dev/dsp. A copy to /dev/dsp is a blocking write, the test buffer is therefore played back at a standard sampling frequency. When the audio test program is started, the creation of the buffer and initialization of the audio output produces a narrow peak in power consumption. The occasional peaks in power consumption during playback are due to the transfer of audio samples from the test program to the audio device driver and the transfer of samples from the driver to the audio chip (Philips UDA1341).

We now consider the power consumption of running the Madplay MP1 decoder on the Assabet. The MP1 format compresses a CD quality audio signal of 1.35 Mbps (16bit, 44.1KHz, stereo) into 384 Kbps without a significant loss in quality. Madplay was used because it is one of the few fixed-point decoders that passed the MPEG audio decoder accuracy tests defined in part 4 of ISO/IEC 11172. A fixed-point implementation is important because the SA1110 processor on the Assabet lacks a

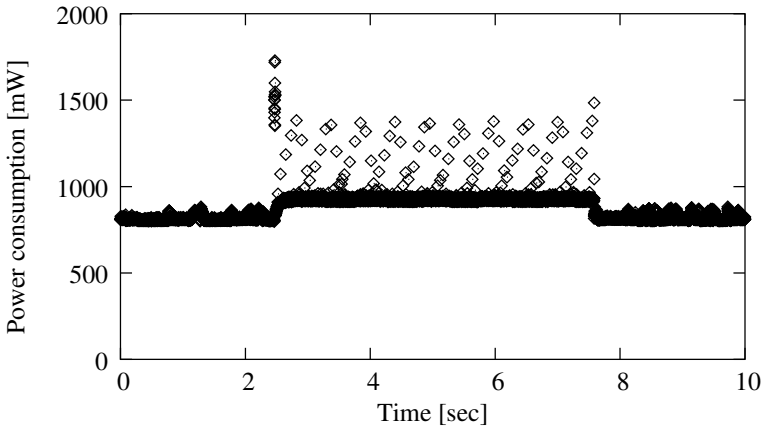


Figure 4.7. Power consumption for audio output on the Assabet.

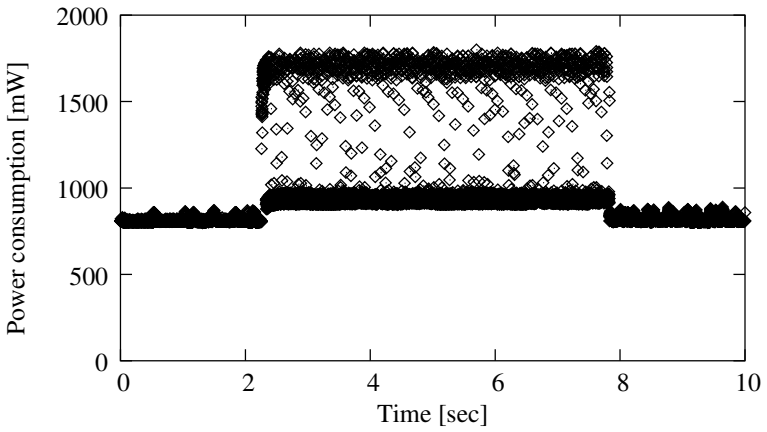


Figure 4.8. Power consumption for MP1 audio decoding.

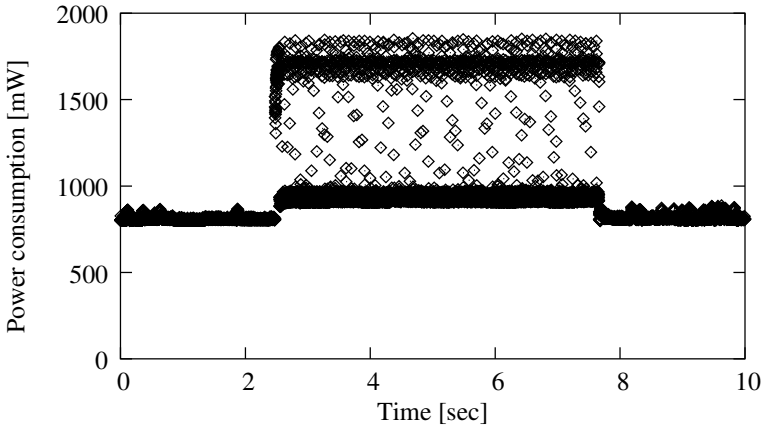


Figure 4.9. Power consumption for MP3 audio decoding.

floating point unit. We used Version 0.14.2b of the publically available Madplay decoder and compiled it for the ARM architecture in the default configuration, without either the CPU optimizations at the cost of audio quality or quality optimizations at the cost of more processing. Madplay can decode MP1, MP2, and MP3 audio files.

Figure 4.8 shows the power consumption of the Madplay MP1 decoder, which varies between almost 1000 mW and 1700 mW. The Madplay decoder uses a compressed audio file stored in the ramdisk of the Assabet. At regular intervals Madplay copies the audio to /dev/dsp. The cost of audio decoding only is equal to the difference between the analog audio output of Figure 4.7 and analog output plus decoding of Figure 4.8. The processor intensive MP1 decoding is clearly visible in Figure 4.8 as the thick black band around 1700 mW. During audio decoding and playback, the Assabet is either consuming almost 1000 mW when playing the buffered content of /dev/dsp or consuming around 1700 mW when decoding a block of compressed audio data. Note that during initialization Madplay uses more power than during normal playback. This is visible in the figure as the thick black band from 1350 up to 1700 mW around 2.2 s.

The Madplay decoder is also capable of decoding the MP3 format efficiently. Figure 4.9 shows the power consumption during MP3 decoding. The decoding of an MP3 file is more processor intensive and therefore consumes more power. Compared to MP1 decoding the thick black band of MP3 decoding has higher peaks and is also shaped differently. The significant differences between the MP1 and MP3 format are therefore also visible in their power consumption patterns.

We measured the power consumption of the Assabet during a long time interval while running Madplay. A summary of the audio decoding costs is shown in Figure 4.10. The idle power consumption (768 mW) of the Assabet was measured at 15 KHz without an attached WLAN card. This is significantly below the previously reported idle power consumption of 1030 mW of the Assabet and WLAN card together.

The audio test program raises the power consumption to 890 mW. The power consumption during 384 Kbps stereo MP1 playback is equal to 1048 mW. MP2 playback at 256 Kbps uses less power than MP1 for the same audio quality (1044 mW). This is counterintuitive, but can be explained by the difference in memory usage and lack of software optimizations for the less popular MP1 format. The widely used MP3 format yields the highest compression ratio and also requires more processing capacity. The MP3 power consumption is 1070 mW at 128 Kbps.

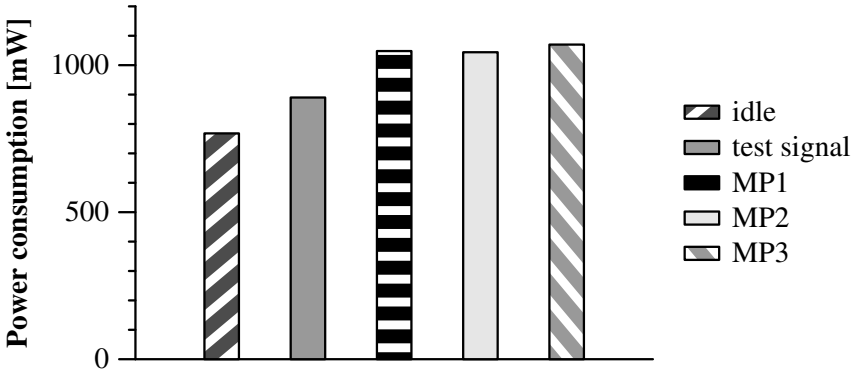


Figure 4.10. Costs of audio decoding on the Assabet.

Format	Power consumption
	mW
MP1	280
MP2	276
MP3	302

Table 4.1. A simple performance model for playback of CD-like quality audio.

Figure 4.10 shows that the power consumption of audio decoding varies little for different compression levels. A simple performance model is sufficient to predict the power consumption. Table 4.1 shows the basic performance model for playback of compressed audio. These numbers are simply the difference between the average power consumption while decoding and the idle power consumption. The cost of analog audio output (122 mW) is included in these figures.

A complete audio decoding performance model is more elaborate due to the trade-off in audio quality and power consumption. Playback of a low-quality audio file consumes less resources than a high-quality file. The average power consumption cost for the playback of a single channel (mono) 32 Kbps MP3 is only 251 mW. At such a low bit rate the audio quality is reduced significantly. Compared to CD-like quality, the power consumption is reduced with 51 mW. Stereo playback involves significant costs, 286 mW at 32 Kbps. Different bit rates influence the power consumption to a lesser extent. MP3 playback with 64 Kbps stereo consumes 295 mW, a difference of 7 mW with 128 Kbps. On average it takes only 5 mW more to decode a very high bit rate MP3 file of 192 Kbps than an MP3 of 128 Kbps.

4.4 Audio streaming

The two performance models presented in the previous sections are used within the audio streaming service. The wireless-link Perfd driver transports the compressed audio from a web server to a local buffer on the Assabet and the audio decoder decompresses this local buffer. The performance models are used to determine the optimal compression level.

This section focuses on the trade-off between components. The wireless-link Perfd driver has several choices for the transfer of the compressed audio. One alternative is to keep the WLAN card activated at all times and to receive just a few packets per second. Another alternative is to exploit a local storage buffer (in internal memory) and switch between using the WLAN card at the maximum

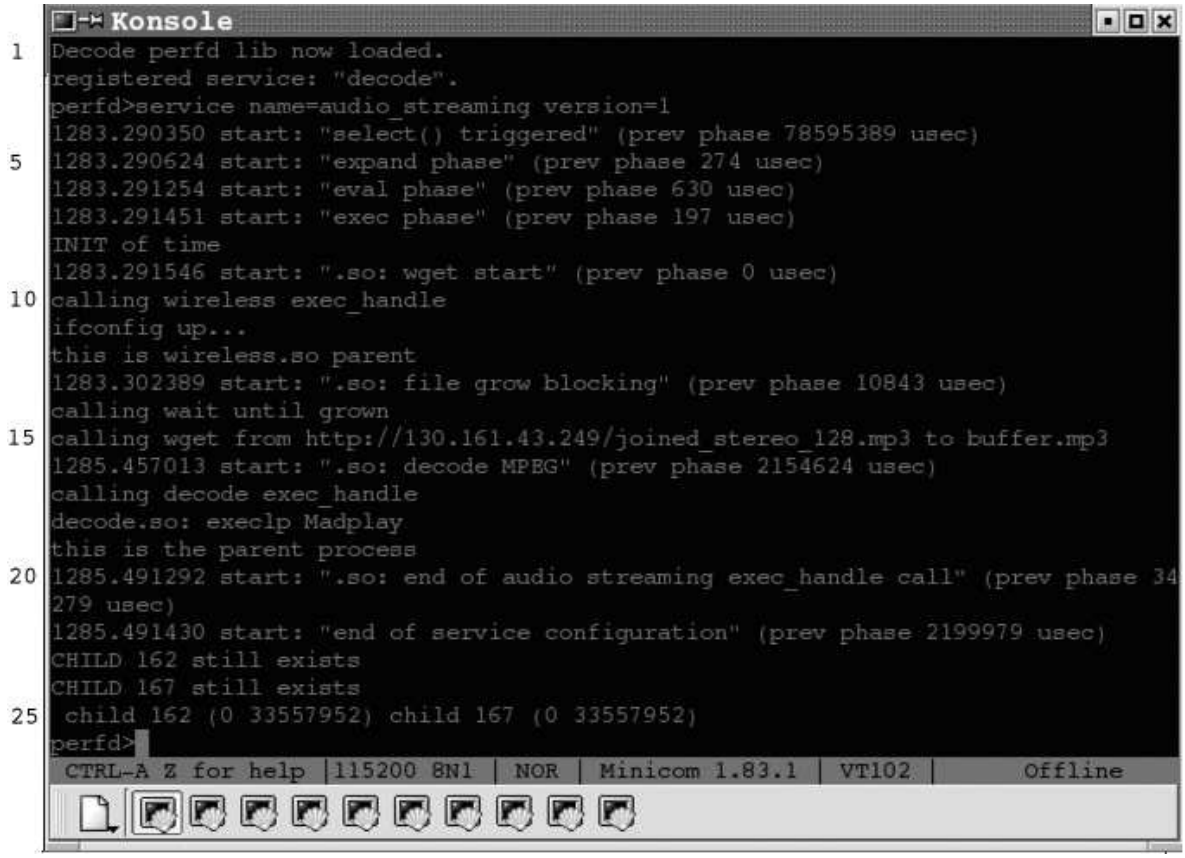


Figure 4.11. A screenshot with debugging information of a Perfd client starting the audio streaming service.

speed and completely deactivating it. This last alternative requires significantly less energy due to the difference in deactivated power and idle power. Activating the WLAN card only once every minute reduces the average power consumption considerably, at the cost of a local buffer (2.81 MB when streaming at 384 Kbps). The wireless-link Perfd driver uses the buffering alternative in our experiments. The maximum size of the local buffer determines the maximum activation interval of the WLAN (e.g. 1 minute). The wireless link cost expressed in terms of energy in the performance model can easily be converted into power consumption by using this activation interval.

In principle it is the task of the Perfd wireless link scheduler to determine the most power-efficient method to exploit deactivated modes (Section 3.2.4). However, for this experiment we did not implement a full Perfd wireless link scheduler. The periodic activation is hard-coded into the Perfd driver.

Figure 4.11 shows a screenshot of the Assabet running Perfd in verbose mode. The first two lines show the loading of the Perfd driver for audio decoding. The third line shows the command starting the audio streaming system with all default options. Subsequent lines, optionally beginning with a time stamp, show various progress reports of the state of the service execution.

After the starting command is entered at the command line client, it is send to the Perfd coordinator. New bytes arriving at the Perfd coordinator command socket trigger the `select()` system call. As shown on line four of Figure 4.11, the `select()` system call is triggered at time stamp

Perfd action	Time
	μsec
initialization	274
expand	630
evaluation	197
total	1101

Table 4.2. Execution times on the Assabet of our Perfd framework implementation.

1283.290350. The time between the detection of new bytes at the Perfd command socket and the start of the configure function in Perfd is 274 μsec. During this time the *service* command is read from the socket, this command is parsed, and a `service_t` structure is filled to be used as an argument for the `configure()` call of the Perfd coordinator.

As shown on lines five and six of Figure 4.11, the expand action starts at time stamp 1283.290624 and ends at 1283.291254. The required time for the expand action of Perfd is therefore 630 μsec. During the expand action of the configuration phase, the streaming driver translates its parameters such as a playlist into parameters for the underlying components such as download URLs on a server. In our experiment a single test track was streamed. For simplicity we assume that a web server is capable of serving all audio tracks at several compression levels by either storing several track versions or conducting real-time transcoding. The server in our experiment stored each track in several quality levels and three compressions levels (MP1, MP2, MP3).

During the evaluation action, power consumption numbers were added to the config-space graph by the evaluation handlers. These handlers use the internal performance models in the Perfd drivers to determine the power consumption of, for example, MP3 decoding with CD-like quality. The duration of the evaluation action was 197 μsec (line 7). Table 4.2 shows the elapsed time for several Perfd actions.

During the execution action, the streaming Perfd driver calls the execution handler of both the wireless link and the decoder (lines 10 and 17). First the wireless link execution handler is called. A process is forked by this execution handler to activate the wireless link (line 11) and to transport the specified audio track (line 15). The activation of the wireless link takes roughly 2 s, as shown in Figure 4.4. The forked process for the wireless link downloads the audio track to a local buffer file on the ramdisk of the Assabet. The streaming driver looks at the size of the local buffer and blocks execution (2.1 s) until the local buffer is filled with a sufficient amount of compressed audio data (line 13). When the local buffer is filled with a few seconds worth of audio data, the execution handler of the decoder is called by the streaming driver. This handler forks a child process that uses the Linux `execlp` system call to execute the Madplay decoder (line 18).

The overhead of the Perfd framework during execution is small. The execution action is not accounted as Perfd overhead because we assume it must always be carried out, in this form or another. Using socket communication is a very general, but not a very fast method of implementing functionality. The Perfd overhead in this example is defined as the time for socket communication, the expand action, and the evaluation action, being 1101 μs.

The overhead of Perfd in this experiment is not limited to these 1101 μs. During the first phase of Perfd deployment a Perfd driver must be developed for every hardware component (Section 3.3.1). In the second phase, a Perfd driver must be crafted for every software and hardware component. For each Perfd driver, initial performance measurements were carried out, the parameters of the performance model were determined, exact measurements were carried out to create a performance model,

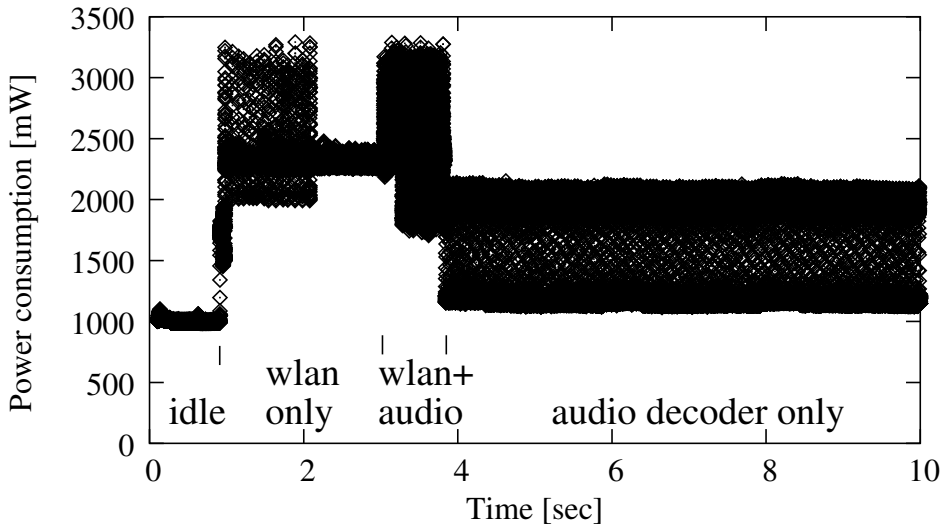


Figure 4.12. The Assabet power consumption for the Perfd audio streaming service.

the performance model was coded in the C programming language, and the Perfd handlers were programmed. Creating a Perfd driver for the wireless link and for the audio decoder took one week each, based on a standard Ph.D. student work week (60 h).

Figure 4.12 shows the power consumption of the Perfd audio streaming service. The plotted interval shows the power consumption of the wireless-link initialization (wlan only), the transfer of audio data (wlan+audio), and the decoding (audio decoding only). To highlight the rapid variations in power consumption the highest sampling rate possible in our measurement setup is used in this figure (15 KHz). The figure with 10 s of data therefore comprises 150,000 points.

The downloading of the compressed audio starts with the sharp increase in power consumption at 3 s from below 2500 mW to above 3000 mW in Figure 4.12. Roughly 10 ms after the downloading started, the Linux CPU scheduler issues a context switch from the downloading child to the streaming Perfd driver that is executing `wait_until_file_has_grown()`. The local buffer is already filled with several seconds worth of compressed audio after these 10 ms, due to the low latency of both the web server and the wireless link Perfd driver, triggering the Madplay playback. As a result the sound becomes audible only just over 10 ms after downloading begins.

The power consumption at the start of the wlan+audio interval ranges from 1600 to 3300 mW. However, during the roughly 100 ms at the start of this interval the power consumption does not drop below 2100 mW. We suspect that this behaviour is due to the start of the Madplay decoder. The decoder uses more power during initialization than during normal playback.

In the final audio streaming experiment of this chapter we compare the power consumption estimations of Perfd and measurements on the Assabet. First we use a wireless link operating in 11 Mbps mode. We will also show results for the 2 Mbps case.

In the experiment we measure the power consumption of the Assabet for 30 s. During the measurement, the audio streaming service is initiated and completed. A test track with a duration of 27 s is used. The 3 s difference leaves sufficient time for the Assabet to return to idle. Another advantage of the 3 s idle time is that no errors are introduced in the measurement due to inaccuracies in measuring the exact start and stop time of the audio streaming service. The problem of accurate measurements

alternative	streaming bit rate	audio decoding	wireless link	wireless cost (interval=27s)	total (including idle)
	Kbps	mW	mJ	mW	mW
MP1	384	280	3559.0	132	1442
MP2	256	276	2808.7	104	1410
MP3	128	302	2058.5	76	1408

Table 4.3. Perfd audio streaming estimations using the audio decoder and wireless link performance models.

alternative	Perfd estimate (27s)	idle (3s)	total estimate (30s)	total measured (30s)
	mW	mW	mW	mW
MP1	1442	1030.0	1401	1390
MP2	1410	1030.0	1372	1397
MP3	1408	1030.0	1370	1388

Table 4.4. Summary of the estimated and measured audio streaming power consumption.

on a portable device is described in detail in [40].

Table 4.3 shows the Perfd estimation of the power consumption for this experiment. The performance model of the audio decoder (Table 4.1) is used for the "audio decoding" column. The wireless link cost in mJ are calculated using the Perfd performance model and the size of the test file. The "wireless cost" column shows the converted wireless link cost in mW, given the length of the audio track. The final column adds the audio decoding, wireless link, and idle power consumption. Note that the MP3 method is estimated to consume the least amount of power, although with only a very small margin. The Perfd coordinator selects the MP3 method as the optimal compression level.

Table 4.4 shows the results of the audio streaming estimations along with the actual measurements on the Assabet. The three rows show the various levels of compression. The MP3 method is automatically selected by the Perfd coordinator because it is estimated to use the least amount of power. The MP1 and MP2 method are selected in our measurements by adding the parameter *force=mp1* and *force=mp2* to the Perfd service request. This optional argument bypasses the selection mechanism and executes the indicated compression level. The "Perfd estimate" column is taken from Table 4.3. The "idle" column lists the average power consumed when the Assabet is idle and the WLAN card is deactivated. The "total estimate" column shows the power consumption estimate for the complete 30 s of the experiment. The "total measured" column shows the actual average power consumption during experiments on the Assabet.

The numbers in Table 4.4 show that the basic performance models are still accurate. The maximum deviation of the total power consumption is 25 mW for the MP2 compression level. This amounts to an error of 1.8 % relative to the total power consumption. This deviation is mainly due to the wireless-link estimation error. Our experience with 802.11 suggests that the addition of one parameter to the performance model would increase its relative accuracy even further. This parameter is the "processor utilization" parameter that reflects the availability of CPU time slots to process incoming HTTP packets. The high load on the CPU due to the MP3 decoding seems to cause an underestimation of the downloading cost. The variability in the time required for the activation of the

alternative	total streaming measured		dynamic power part		relative dynamic power difference	
	11 Mbps	2 Mbps	11 Mbps	2 Mbps	11 Mbps	2 Mbps
	mW	mW	mW	mW	%	%
MP1	1390	1543	360	513	98.0	100.0
MP2	1397	1465	367	435	100.0	84.8
MP3	1388	1421	358	391	97.7	76.2

Table 4.5. The audio streaming power consumption in both 11 Mbps and 2 Mbps mode.

WLAN card also introduces small errors.

Our three cooperating Perfd drivers are already capable of predicting the power consumption resonably accurately by using the most basic performance models. The config-space graph has proven to be capable of making alternatives explicit in our experiment. The highest total error of the total power consumption for one alternative is only 1.8 %. This number is distorted due to the exceptionally high idle power consumption of our experimental platform. For more elaborate performance models on platforms of commercial quality we expect that it is possible to obtain a power consumption error of less than 5 %. If acurate performance models are viable, the Perfd framework would be capable of estimating the remaining battery lifetime in advance. Current estimators of remaining battery lifetime on laptops are based on extrapolation of current usage patterns and measurement of battery voltage. By exploiting performance models, Perfd carefully avoids the inaccurate method of extrapolating the current power consumption and complex modeling of chemical battery processes [144].

The audio streaming experiment parameters in Table 4.4, CD-like quality and 11 Mbps mode WLAN, are a special case. In this case the cost of the wireless link and the decoder were almost balanced. The three compression levels use almost the same amount of power. The difference in power consumption between MP1 and MP3 is a mere 2 mW. Due to the high idle power consumption of the Assabet the *relative* difference is marginal (0.14 %).

This experiment shows that Perfd has the ability to predict power consumption and is able to choose between alternatives. However, no real savings are obtained. Therefore, we have extended the previous streaming experiment and demonstrate that actual power savings can be obtained. In this extension, we force the WLAN card in 2 Mbps mode.

When the receiver-transmitter distance is increased and the WLAN card is forced to operate in 2 Mbps mode, the difference in power consumption between the compression levels would be more pronounced. With the significantly increased cost of the wireless link, the MP3 compression level will be more beneficial (Figure 4.6). On the other hand, changes in the audio quality from CD-like to AM-radio is expected to yield significant differences, not in favor of MP3. For MP3 compression the decoding would cost 51 mW less than CD-quality, but the wireless-link cost would decrease substantially if the test file size were reduced with a factor of 4 and would no longer be a dominant factor. Less compression such as MP1 will be more beneficial.

Table 4.5 compares the results of the 2 Mbps and the 11 Mbps audio streaming experiment. The three rows again show the three levels of compression. The first column shows the average power consumption of the Assabet and WLAN (in 11 Mbps mode), identical to the last column of Figure 4.4. The second column shows the average power consumption when the WLAN was forced to operate in 2 Mbps mode. The increased cost of the wireless link is the least visible in the case with the highest compression level. The power consumption for MP3 streaming is increased by 33 mW, compared to the 11 Mbps case. For MP2 streaming the average power consumption is increased by 68 mW; MP1

	54 Mbps mode	11 Mbps mode	2 Mbps mode
CD-like quality	MP2 (344mW)	MP1/MP3 (378mW)	MP3 (401mW)
low quality	MP2 (277mW)	MP2 (286mW)	MP3 (348mW)

Table 4.6. The optimal compression level and estimated power consumption for different settings.

streaming by 153 mW.

The high Assabet idle power consumption is unrealistic for a commercial product. The lack of support for a WLAN card on the LART platform cannot, by itself, explain the 978 mW difference in idle power consumption between the Assabet and the LART. We therefore focus on the "dynamic power" and omit the Assabet static idle power consumption (1030.0 mW). The third and fourth column state the average dynamic power consumption for both experiments.

The dynamic power consumption varies only slightly for the three compression levels in the 11 Mbps audio streaming experiment. The relative difference is limited to just a few percent. The last two columns show the relative difference, with the highest power consumption level set at 100 %. These last columns show the relative difference between the three compression levels. The 2 Mbps case shows that 23.8 % less dynamic power is consumed when the MP3 compression level is used instead of the MP1 one.

The audio streaming experiments show that Perfd enables the prediction of power consumption using basic performance models. Perfd increases the power efficiency by predicting the power consumed by various operational modes, such as the compression level. Measurements show that Perfd correcly selects the MP3 compression level, yielding a 23.8 % reduction in dynamic power consumption versus the MP1 compression level.

Besides the 2 and 11 Mbps case, we can also consider a 54 Mbps mode when we take into account the newly developed hardware based on the 802.11g standard. The 802.11g standard improves the 802.11b standard by using OFDM modulation instead of direct sequence spread spectrum (Section 2.1.3). The result is a higher throughput in the same frequency band. During the first months of 2003, the first operational 802.11g prototypes were demonstrated.

We now speculate on the effect of using a 54 Mbps mode and modifying the audio quality in the streaming experiment. Because 802.11g hardware is currently not available, we have to estimate the performance model. The preliminary documentation of an early 802.11g product (Atheros AR5001X) does not contain sufficient details [4]. Measurements on another 802.11 product that uses OFDM (Atheros AR5000) showed that the energy usage roughly scales with the link speed. We assume the following performance model for 802.11g: $E_{54Mbps} = 1310 + \frac{11}{54} \cdot 1610 \cdot S_{file}$. This model is an extrapolation of the 11 Mbps model.

Table 4.6 shows the optimal compression level for two quality levels and three wireless link modes. This table is based on previous measurements and estimations for the dynamic power consumption and performance of 802.11g products. The "low quality" settings are based on bit rates that are reduced by a factor of four, compared to "CD-like quality". The power consumption of the Madplay decoder will be reduced due to the lowered bit rate (Section 4.3). For example, the dynamic power consumption of MP2 audio streaming in 54 Mbps mode is estimated to be 277 mW (224mW for 64Kbps MP2, 53mW for 802.11g).

The proximity of a base station and the environment determine the mode of a wireless link. In 802.11g products this mode ranges from 1 to 54 Mbps. The difference in dynamic power consumption in Table 4.6 shows that changes in the operational mode of the wireless link must trigger changes in

the audio streaming (e.g. compression level) in order to minimize the power consumption. Perfd provides a general-purpose infrastructure for such adaptive systems.

4.5 Conclusions

In this chapter we have demonstrated the ability of Perfd to make trade-offs at execution time. Two aspects of trade-offs that were previously mostly isolated research fields are now unified by Perfd within a single framework. Classical research on ”QoS” is focussed on the negotiation process and trade-offs *before* service delivery. Adaptive systems research is focussed on trade-offs *during* service delivery.

The overhead of our Perfd implementation during execution time is limited and will most likely be acceptable (1.1 ms in our experiment). The implementation overhead due to the requirement of constructing extra Perfd drivers is also acceptable. When compared to the development time of OS device drivers, the development of a Perfd driver is far less time consuming. With the growing demand for smaller, yet more powerful portable devices, the investment in Perfd drivers seems a price that is worth paying. As described in Section 3.1.1, the improvements in power efficiency of a power management solution must outweigh the implementation cost.

The research community and the market will ultimately determine if the Perfd framework is either overly complex, provides too little gain, or is a usable solution. As stated before, many researchers have recognized the potential of cooperative power management and the lack of general framework to realize it [16; 17; 55; 139]. We regard the Perfd framework as a first realization of cooperative power management.

Chapter 5

Scheduling

THIS chapter describes a detailed experiment for the scheduling phase of the Perfd framework. The goal is to evaluate the Perfd framework on a fully operational portable device. This evaluation provides actual Perfd power-efficiency improvements for the specific problem of clock scheduling on a general-purpose CPU.

Roadmap

Recall that cooperating power-management policies are at the heart of Perfd. To exploit the voltage scaling mechanism of the processing component we developed a cooperating clock-scheduling policy called *PowerScale*. A cooperative policy requires both context awareness and access to information (Figure 1.6). PowerScale is cooperative and therefore: understands its environment, interacts, exchanges information, and can estimate the future. PowerScale is implemented as a Perfd driver on our LART platform and controls the voltage scaling mechanism described in Section 2.1.2.

5.1 Clock scheduling

A clock scheduler optimizes the processor frequency with respect to the workload to be serviced. A clock scheduler must determine which task to run, when the clock frequency needs to be changed and to what frequency. This problem, known as *clock scheduling*, is the central problem addressed in this chapter. Creating efficient clock scheduling policies is considered a difficult problem. In the last decade researchers have proposed various approaches to solve this problem with limited success. See [67] for an elaborate introduction to the clock scheduling problem.

5.1.1 Architecture

The placement of a clock scheduling policy in the architecture (e.g. hardware versus OS level) has a significant impact on power efficiency (Section 2.4.1: Policy placement). As shown in Figure 3.2, the Perfd framework uses a distributed policy. Our cooperative approach requires a processing configurator in the higher layered software.

A clock scheduling policy needs to include more intelligence than a policy that uses a simple "power down on idle". The policy needs to output a *performance level* for the processor (Section 2.3.3: mechanism control resolution). The performance level of the processor has a direct impact on both the performance of the portable device and the battery lifetime. The processor usually only supports a small number of clock frequencies. A workload misprediction of just a few percent will result in a clock frequency setting that is either too low or too high.

For a fluctuating workload, a clock scheduler will often increase or decrease the clock frequency too late. The optimal clock schedule, which minimizes energy usage, defines for each point in time which task to run at a specified frequency. Without detailed a-priori information on the workload it is *impossible* for a clock scheduling policy to determine the optimal clock schedule. The various proposed policies differ significantly in their success to *approximate* the optimal schedule. For example, the excellent research presented in [145] showed power consumption results for a fluctuating video decoding workload. Their simulations of a policy produced a clock schedule that was 36 % above the optimum. Simple clock scheduling policies mispredict the optimal frequency often and produce only modest power savings, while advanced policies may yield CPU power savings of up to 50 % [152].

Figure 5.1 shows both the Perfd framework view (a) and the implementation view (b) of our PowerScale clock scheduler. In the Perfd view, the configurators of several components send reservations to the processing components. A performance model of the processing service is also available. Reservations are processed by PowerScale to form a power-efficient access schedule for the processing service.

Figure 5.1.b shows the details of the implementation. The actual switch of the processor clock frequency is handled by an OS device driver. We have implemented such a driver for the Linux OS (details can be found in [151]). Our OS device driver adapts the memory and bus configurations and is the first to also adjust the supply voltage of the processor at run time. Note that to remain power efficient a change in frequency implies a corresponding change in supply voltage. Our LART device driver has a hard-coded table of the frequency and voltage combinations, derived from Figure 2.3. The device driver sends the voltage setting to a D/A converter. This D/A converter in turn is connected to a voltage-controlled DC/DC converter. On our LART platform, the switch of frequency and voltage takes 140 μs, during which the system is stalled. This implies that settings can be changed frequently without causing too much overhead.

In this chapter we argue that clock scheduling in a general-purpose context can only be effective when software components (applications) cooperate. It is vital that software communicates its (fu-

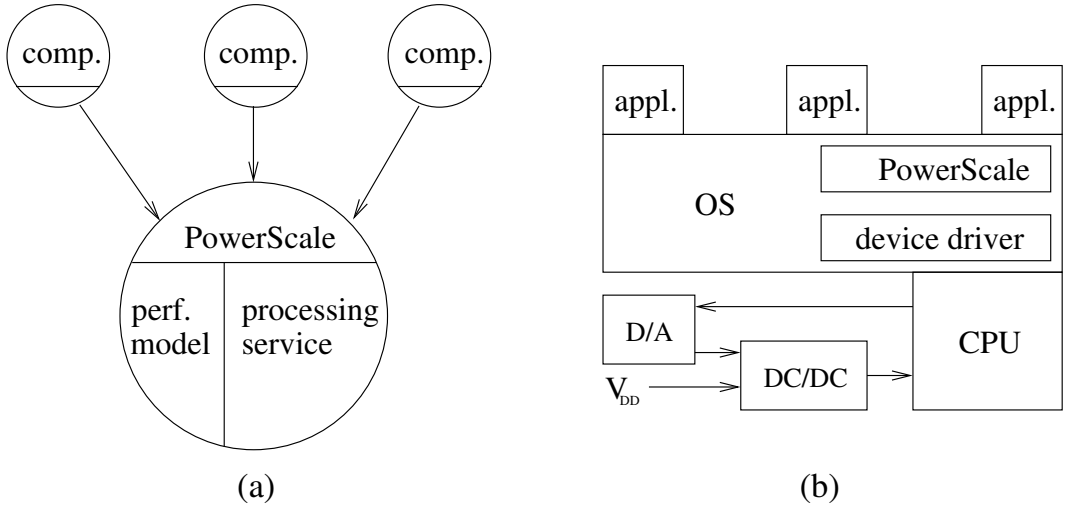


Figure 5.1. The Perfd (a) and implementation (b) view of the PowerScale clock scheduler.

	dedicated hardware	compiler	real-time OS		general-purpose OS			cooperative
			fixed	dynamic	hardw.	interval	integr.	
Quality	++	–	+/-	+	–	-	+/-	++
Availability	design	compile	design	run	run	run	run	run

Table 5.1. Comparison of clock scheduling approaches in several areas.

ture) processing needs, much like in real-time systems. Only then the clock scheduler can handle bursty (unpredictable) applications and compute an optimal schedule. The general clock scheduling problem itself is NP complete. We present a new heuristic scheduling algorithm called *energy priority scheduling* (EPS) that uses workload descriptions to compute energy-efficient schedules. We have implemented the algorithm as part of the Linux OS and performed several experiments on our variable-voltage LART platform. In particular we demonstrate its ability to schedule a computational task with a bursty video playback task; the computational task is executed between two low-complexity video frames.

5.1.2 Approaches and related work

Clock scheduling has been investigated in the context of four main areas: dedicated hardware, compilers, real-time OSs, and general-purpose OSs. These areas differ in the time of fixation of the clock schedule, the time at which scheduling information is available, and the amount and quality of that information. With dedicated hardware, the clock schedule is determined at design time, using a priori information derived from the application. A compiler, in contrast, can only extract a limited amount of information from the source code of an application to determine the clock schedule. A real-time OS includes a task scheduler that takes into account start times, deadlines, and required cycles, allowing more flexible clock scheduling schemes. A general-purpose OS has to derive a clock schedule from run-time statistics, such as the processor utilization in previous periods. In general, clock scheduling becomes simpler and more power efficient when more (accurate) information is available. An overview of the different approaches is shown in Table 5.1. The first row lists the quality of the information that is available to solve the clock scheduling problem, ranging from very poor (- -) to very

good (++). The second row lists at what time workload information is available: during design time, compile time, or at run time. In the sequel, we will use the approaches in Table 5.1 to discuss related work.

Dedicated hardware

When dedicated hardware is created, for example, a GSM speech codec or JPEG compressor, all possible workload details are known in advance. Therefore the *optimal* clock schedule can often be calculated with brute force at chip design time [116]. This can be costly since the non-preemptive clock scheduling problem, where a task cannot be interrupted, is NP complete [83]; Hong et al. present an effective heuristic yielding schedules that are within 2 % of the optimum [83]. In the preemptive case the optimal schedule can be computed with an $O(n \log^2 n)$ off-line algorithm, where n equals the number of tasks to be scheduled [194].

Compiler

When a compiler is used to determine the clock schedule, the greatest problem is deducing the appropriate information from the source code [6]. For example, deriving the execution time on the target platform from the high-level program code is a non-trivial task. This forces the compiler to make conservative assumptions and yields low-quality scheduling information. If extensive profiling information is present, the scheduling techniques for dedicated hardware can be used. Otherwise, heuristics must be applied to identify code sections that can be executed at low speeds. For example, Hsu et al. describe a system that is based on identifying memory-bound loops [84]. Within such loops the clock frequency can be reduced, since the memory subsystem is much slower than the processor. However, this approach is only effective when memory-bound loops occur frequently and the cost of a frequency/voltage change is negligible relative to the total execution time of a loop. A hybrid example is proposed in [6] were the compiler generates "program checkpoints" at which the frequency of the clock can be adjusted. Such program checkpoints are pieces of generated code that re-calculate the clock frequency.

Real-time OS

In the realm of real-time operating systems, voltage scaling focuses on minimizing power consumption of the system, while still meeting strict task deadlines. Real-time tasks specify their starting times and deadlines; tasks that must be repeated also specify their periods. In hard real-time systems the worst-case execution time (WCET) can often be obtained at software design time through static analysis, profiling, or direct measurements [77; 115]. When all details of the workload are known and the schedulability is verified at design time we classify such systems under "fixed real time". When details of the workload, such as WCET, or even the tasks themselves are only known at run time we classify such systems under "dynamic real time". For example, for multimedia servers, the exact workload is only available at run time [38]. An admission controller needs to determine if new tasks can be scheduled and admitted.

An example of a scheduler for fixed real-time systems is the *Average Rate* run-time heuristic by Yao et al., which has been proved to consume at most a factor of 8 more energy than the optimal preemptive schedule [194]. Pering et al. present a dynamic real-time system based on *Earliest Deadline First* (EDF) scheduling [146]. They assume that tasks specify no start times and, hence, can be executed at any moment. Measurements show that significant energy savings can be obtained (20 % of peak power) for some applications.

In both classes of real-time OSes the WCET is used to check the schedulability and possibilities for reducing the clock frequency in the schedule without violating deadlines. For example, the algorithm in [83] initially schedules all tasks at maximum frequency. After that the task schedule is adjusted until no further reduction is possible without violating deadlines. The ratio between the actual execution time and the WCET can be quite low: an average of 0.5 is reported for several hard real-time applications studied in [57]. When the WCET is not an accurate estimation of the execution time, the assigned clock frequencies to meet deadlines tend to be too high (factor of 2 on average). Consequently, a task usually finishes early, and an idle periods occurs. If another task is eligible for execution, however, the idle period can be used to execute that task at a reduced speed, see [169].

A recent paper by Pillai et al. discusses an alternative approach to handle the conservative WCET rarely encountered in practice [148]. Their "look-ahead" clock scheduling algorithm is based on an EDF scheduler. The task with the earliest deadline is scheduled with the lowest possible processor speed that does not violate its deadline. As a result other tasks must be scheduled at a high frequency to compensate. The assumption is that the task is not likely to use its WCET and will finish early. When the task finishes early, energy is saved and the next task with the earliest deadline is scheduled at its lowest possible frequency. Note that for some (artificial) workloads the look-ahead algorithm may defer tasks too aggressively and actually increase power consumption, as can be derived from their simulation results. Unfortunately they do not provide insight in how well their heuristic performs in comparison to the optimal schedules. Given that they do not preempt tasks, their scheduler is limited in its possibilities.

General-purpose OS

Clock scheduling in the context of a general-purpose OS is difficult, since little information is known about the applications. Applications do not communicate deadlines or priorities to the OS, hence, all the clock scheduler can do is observe the load that has been generated in the past and extrapolate into the future. The clock scheduler measures the processor load in fixed intervals, for example, every 20 ms. A common technique is to use two boundary values on the processor load to decide whether to increase, decrease, or keep the current clock frequency in the next interval. If the measured processor load drops below the lower bound, the processor frequency is decreased. Similarly, if the processor load rises above the upper bound, the frequency is increased. This technique is called interval-based clock scheduling, or interval scheduling in short.

The Transmeta Crusoe processor is a prime example of a clock scheduling policy located at the lowest possible level on a computer running a general-purpose OS. The Crusoe processor has built-in support for clock scheduling in the "microcode" of the processor [181]. Unfortunately, little information is made available about the exact workings of the "LongRun" technology, but it is clear that it can operate in isolation, that is, without any help from the OS or application [63]. The microcode has only a little awareness of the global system state, for example, it cannot distinguish OS foreground tasks from background tasks.

Weiser et al. first presented the idea of interval-based voltage scaling for a general-purpose OS in 1994 [186]. Most contributions regarding interval-based voltage scaling consist of theoretical analysis [92] and simulations [66; 112; 145]. The simulation studies show that interval scheduling reduces power consumption considerably compared to running a system at full power. There are, however, some fundamental problems. First, the optimal interval length is application dependent. Second, bursty applications with unpredictable workloads cannot be scheduled effectively at all. The simulations by Pering et al. show that the power consumption of their interval schedule for video decoding was 36 % above the optimum. Recent measurements on actual hardware by Grunwald et al. confirm

these observations [68].

Traditional interval scheduling based on the processor load can be improved by incorporating other information (run-time statistics) to estimate the processing requirements of applications. Such "integrated clock schedulers" require numerous modifications to the OS. For example, Flautner et al. [59] describe an integrated scheduler that maintains processor usage statistics of every process, observes the communication pattern between processes, keeps track of input/output device usage by processes, and tries to extract deadlines from periodic tasks. The simulations results are promising, but a comparison with a traditional interval scheduler is not included, so the advantage of using additional information is not known. Another aspect that needs additional study is the effect of their scheduler on bursty and cpu-intensive applications, such as video decoding and speech recognition, because such applications were not included in the simulations.

User-related timing characteristics are another useful source of information for integrated clock scheduling [59; 120]. For example, Lorch et al. [120] exploit the observation that a reaction time of 50 ms for interactive applications is below the perception threshold of the user. Therefore, the application processing time for, say, a mouse click can be increased to 50 ms (by slowing down the CPU) without noticeable performance degradation. Off-line simulations show that the upper bound on the additional energy saving is in the order of 20 %. It remains to be seen how much energy can actually be saved in a real implementation. It will be difficult for an integrated clock scheduler to "see" the dependencies of executing threads. For example, N threads that update several windows on the screen can only be delayed *together* for a total time of 50 ms without noticeable performance degradation, instead of for $50N$ ms.

Cooperative clock scheduling

The Perfd framework is designed to enable cooperation. Cooperation means involving applications in the clock scheduling problem. Reliable, accurate information for solving the clock scheduling problem can only be obtained from the applications themselves. When applications operating on a general-purpose OS are modified to register their processing requirements (cycles, deadlines, etc), clock scheduling becomes simpler and more effective. Cooperative clock scheduling holds two opportunities for further power savings compared to using the integrated clock scheduler. The first opportunity is the possibility of updating the processing requirements. The second opportunity is to use average processing requirements instead of the worst-case estimates that are used in real-time systems.

We call the updating of task processing requirements in cooperative scheduling *intra-task information updates*. Intra-task information updates are proposed in a recent article by Shin et al. [168]. They combine the compiler-based approach with cooperative clock scheduling. Shin et al. use source code analysis to extract the WCET and combine it with a run-time component. An MPEG 4 decoder is used as a case study for their analysis tool. The tool calculates off-line the WCET updates for several points in the MPEG 4 frame decoding process. For example, at the start of the frame decoding process, only the overall worst WCET of any frame type is known. When the frame type is determined, it is replaced by the WCET of that frame type. During the decoding of the frame even more information becomes available (such as motion vectors and macroblocks) and the WCET is updated and converges to the actual frame decoding time. This refinement technique is guaranteed to meet hard real-time deadline requirements (i.e. frame deadlines). The disadvantage is that the overall WCET is not very likely to occur and consequently, the clock speed is set far too high at the beginning of every frame.

In contrast to real-time systems, applications operating on a general-purpose OS are not time crit-

ical and deadlines may occasionally be missed. Typical laptop applications such as word processing, games, video editing, and (wireless) web browsing are real-time (interactive) applications, yet their deadlines are soft. Users will tolerate (some) jitter in response times. This allows for an easy solution to the problems associated with the WCET. Applications may report their average execution time (AET), which is generally a much better estimator of the true execution time, leading to more power-efficient clock schedules. This is the approach we take in PowerScale. Note the resemblance with the look-ahead algorithm from Pillai [148] (see Section 5.1.2). The look-ahead algorithm is the only algorithm known to us that also exploits the low likelihood of WCET occurrences *before* the actual execution of a task.

The cooperative clock scheduling algorithm presented in this chapter can be applied in both a real-time and general-purpose OS context. The applications on a general-purpose OS need to be modified to pass on intra-task information updates and to indicate their AET.

5.2 Video decoding

To exploit the power consumption reduction of clock scheduling, we propose to make applications cooperative such that bursty and cpu-intensive applications can decrease their power consumption by indicating their processor usage to the clock scheduler. We modified a video decoder to estimate its average execution time (AET) for each frame and communicate this requirement along with the frame deadline to the clock scheduler. In this section, we briefly discuss H.263 video compression, our method for estimating the frame decoding time, and our modified H.263 application.

5.2.1 H.263 video compression

The H.263 standard is created for low-bitrate video compression [158]. The standard is based on both H.261 and MPEG2. H.263 frames are displayed at a fixed rate. Throughout this chapter we use a frame rate of 15 frames per second, which means a maximum decoding time of 67 ms per frame. H.263 defines three types of frames: I-frames (intra picture), P-frames (predicted picture), and B-frames (bidirectional predicted picture). I-frames are self-contained images, similar to JPEG. P-frames encode the difference from a previous I or P frame. B-frames contain references to both the previous and the succeeding frame. Because a B-frame contains forward references, the succeeding frame must be decoded prior to the B-frame itself. As a result the decoder must process two frames in a single frame time. We use the PB-frame notation to indicate the frame in which two dependent consecutive frames are decoded.

A frame consists of a grid of blocks that measure 16x16 pixels, called macroblocks. A macroblock in a P-frame consists of the differences with a reference to the previous frame. The macroblock is displaced by a vector to compensate for motion. Motion compensation is used to decrease the difference from the previous frame. This yields a high video compression level, as only a minimal amount of information is encoded. The pixels in each macroblock are efficiently encoded using a Discrete Cosine Transform (DCT), which is a computationally intensive operation. The number of bytes for each macroblock in the encoder output is variable. Macroblocks that contain no information are not inserted into the compressed bit stream. The number of inverse DCTs required to decode the macroblocks is therefore variable. This is the main cause of the bursty behavior of both H.263 and MPEG2 decoders. Variable-length encoding (e.g. Huffman coding) is used to further compress the macroblocks, the motion vectors, and the frame header. Note that the type of the frame is known to the decoder only *after* the variable-length decoding step.

5.2.2 AET estimation

It is difficult to predict the decoding times due to the wide variation in scenes (e.g. talking heads versus MTV). A frame that is very similar to its predecessor requires few encoded macroblocks to capture the difference and hence takes little time to decode. Frames that differ considerably from their predecessor result in longer decoding times. Figure 5.2 shows the variation of the frame size for the standard Carphone test sequence (190 frames), which was encoded using the Telenor H.263 encoder V2.0 with the following settings: qcif resolution, 15 fps, default quantization, unrestricted motion vectors, syntax-based arithmetic coding, advanced prediction mode, and use of PB-frames. Note the large initial I-frame in the upper left corner.

Various methods have been developed to estimate the AET of a video frame. One method is to include a complete reference decoder inside the encoder and to measure the actual frame decoding times. These decoding times are added to the compressed video sequence. This method is proposed in [25], but requires a reference decoder for each target platform. This drawback can be eliminated by using a generic model of the frame decoding complexity. Such a complexity model for MPEG4 (7 parameters) is presented in [131]. They report accurate results (error ≤ 5 %). The drawback of their method, however, is the necessity to modify MPEG4 encoders to include the complexity parameters in each frame header.

To ensure backward compatibility and general acceptance, one must avoid modification of video sequences (to include clock scheduling information) whenever possible. Therefore it is interesting to find out which property from the H.263 frame gives a good estimation of the AET of the frame decoding process and can be obtained without adding *any* knowledge to the H.263 compressed bit stream. We found that the combination of frame type and frame size yields an estimation that is simple, yet accurate. A similar estimator for MPEG2 is presented in [15].

Figure 5.3 plots the decoding time versus the frame size for the Carphone test sequence on the LART platform. Two frequencies are used to decode frames of both the P and PB type. The figure shows that video decoding is a demanding application: at the lowest clock frequency of our LART platform, none of the frames can be decoded within the required 67 ms (i.e. 15 frames per second). Furthermore, running the processor on a high speed (221 MHz) is only necessary for the largest PB-frames. The cost of decoding a two-image PB-frame is consistently higher than that of decoding a single-image P-frame. Simple P-frames decode in roughly 75 ms at the lowest clock frequency; the most complex PB-frames take almost 200 ms to decode at this speed, a significant difference. Measurements on more test sequences show that frame decoding times are independent of the content of the test sequence itself; they only depend on the frame type and length [149]. Note that changing the spatial or temporal resolution keeps the linear relation between frame size and decoding time, but modifies the parameters. Fortunately such resolution changes do not occur inside normal video sequences. The characteristics of Figure 5.3 will be used to estimate the minimal processing requirements for each frame.

The type of the video frame is indicated in the frame header. Unfortunately, the frame length is not part of the header, so we cannot directly determine the most suitable AET. The frame length can optionally be added to the header by using the H.263 PEI header field. This requires changing H.263 encoders to add the frame length information to the header. A solution that modifies only the decoder is preferred. By using input buffering in the decoder it would also be possible to determine the frame length before commencing with the decoding. However, this approach would increase the decoding latency, which is a severe drawback for interactive applications such as video conferencing and would also increase power consumption due to the increased memory requirements. We implemented three solutions for this frame length problem, described in the next section.

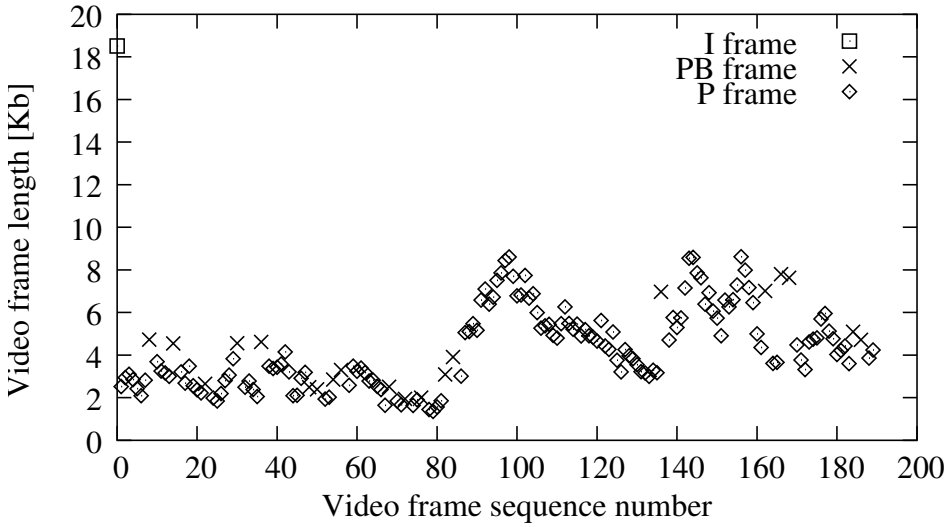


Figure 5.2. Frame size variation over time.

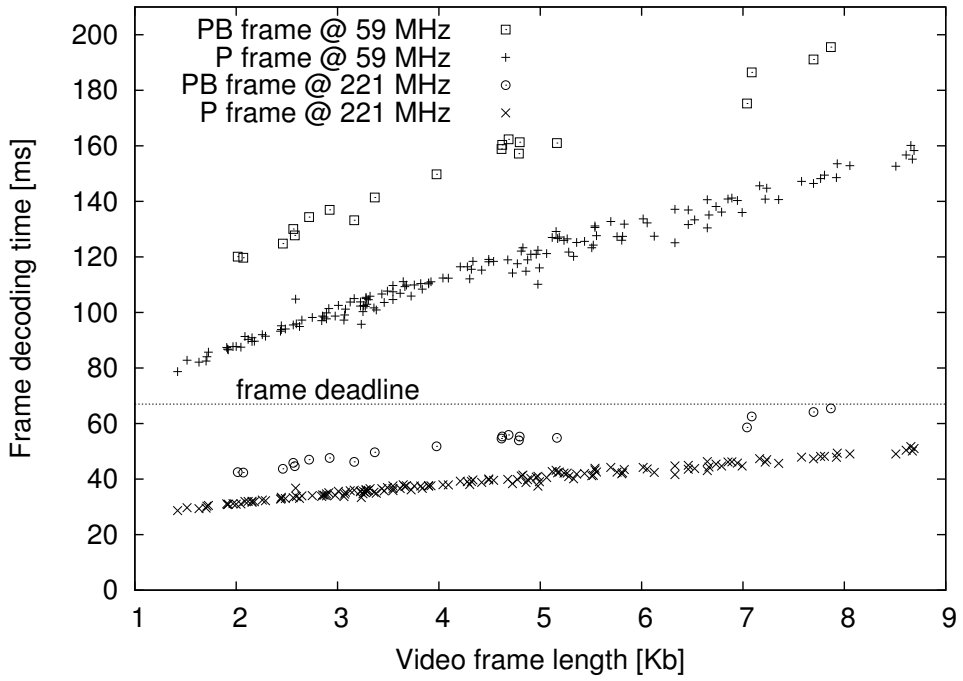


Figure 5.3. Decoding time vs. frame size and type.

5.2.3 Implementation

We extended the Telenor H.263 decoder with an AET estimator based on the observed linear relation between frame size and clock frequency for equal frame types (Figure 5.3). Three modes are supported by our enhancement: *optimal*, *feed forward*, and *feed backward*. The difference between the three modes is the knowledge about the frame decoding times. The optimal mode uses a priori knowledge about the decoding requirements. Through off-line analysis the exact processing requirements are determined for each video frame and made available to the decoder, similar to [25]. This mode provides a lower bound in terms of power consumption for the clock schedule. The feed forward mode uses a priori knowledge about the frame length through the PEI header field. The feed backward mode does not require any modifications to the encoder nor the compressed bit stream, and uses intra-task information updates to adjust mispredicted AETs. Our H.263 decoder communicates its processing requirements for the next deadline with the EPS scheduler once per frame in both feed forward and optimal mode and twice in feed backward mode.

The power-aware decoder in feed backward mode uses several heuristics to estimate the processing requirements as accurately as possible. Frame statistics are kept for each frame type. Initially the decoder requests the maximum processing capacity for the first few frames of a sequence, until an estimation of the time-framelength relation becomes available (using a least squares fit). The best speed for the previous P-frame is used as a starting point for the current P-frame. When the upper half of the video frame is decoded an intra-task update is calculated and sent to the clock scheduler. The decoding time and size of the first 50 % of the macroblocks is used to determine the decoding progress. The remaining 50 % of the macroblocks must also be decoded in the same frame time. When the decoder is running ahead or behind, the estimated time-framelength relation is used to calculate the new processing requirements. Unfortunately, the complexity of the upper half of the image of a video frame is not always equal to the bottom half. To compensate for this, we use the complexity ratio of the upper and lower half of the image from the previous frame to update the AET. We thereby assume that the complexity ratio is a slowly changing parameter in a video sequence.

At the start of a frame the speed of the previous frame is only maintained if there is no frame type change. When a PB-frame follows a P-frame, the speed of the most recent PB-frame is used because generally the previous P-frame has a lower processing requirement.

The "frame_type_len" estimator from [15] also uses the type and length of the previous decoded frames to estimate the decoding time of the current frame. For their calculations they require the off-line calculated relation between frame size and decoding time. Our implementation is similar to the frame_type_len estimator, but we created an on-line version that uses intra-task information updates.

Our power-aware video decoder induces no loss of generality, since it does not depend on H.263 specifics. Our enhancements can therefore also be applied to other compression formats (MPEG2, MPEG4).

The usage of estimates entails the risk of missing the final frame deadline. The chance can be reduced by partitioning the frame into N parts instead of just two, for example, in [168] $N = 33$. We will show that with the most simple case of $N = 2$, the power consumption of the decoder in feed backward mode is already close to that of the optimal mode (see Section 5.4.2). The next section describes a clock scheduling algorithm that combines the processing requirements of, for example, the power-aware video decoder with other applications.

		case 1	case 2
	e_j	$s_j - d_j$	$s_j - d_j$
A	2	$0-3$	$0-3$
B	2	$0-6$	$0-6$
C	1		$4-6$

Table 5.2. Workload descriptions.

5.3 Processing

The processing service in Perfd requires cooperation to determine an efficient clock schedule. In cooperative clock scheduling, applications specify their AET to the next deadline and use intra-task information updates to increase the power efficiency of the clock schedule. Our *energy priority scheduler* is an incremental on-line algorithm that dynamically adjusts the clock schedule within PowerScale when a new task enters the system or an old task completes its execution.

5.3.1 Model

This section defines a model for clock scheduling. The model combines and enhances the models presented in [194] and [129]. Each real-time task j is defined by:

- s_j Starting time
- d_j Deadline time
- e_j Execution time at highest speed

The execution interval of task j is $[s_j, d_j]$. The energy priority scheduling algorithm is used to determine:

- $s(t)$ Speed of the processor at time t
- $run(t)$ Task that is executed on the processor at time t

We further define the following parameters:

- $N_j(tr)$ Number of tasks overlapping with time region tr besides task j
- $N_j = \sum_{tr \subseteq [s_j, d_j]} \frac{\|tr\|}{d_j - s_j} N_j(tr)$ Average number of other tasks besides task j
- $f_j = \frac{e_j}{d_j - s_j}$ Flat processor rate of task j, using the least amount of energy
- $u_j =$ The processor utilization currently scheduled in time region tr_j

5.3.2 Algorithm

The problem is how to schedule a number of real-time tasks j at the lowest possible processor speed that does not violate any deadline.

Before describing our algorithm, we first present two examples that motivate the scheduling heuristic we employ. Table 5.2 gives two simple workloads. The first case consists of just two tasks (A and B). An incremental scheduler considers the tasks one by one. Following the *Average Rate* heuristic by Yao et. al. [194], we simply add the minimum required flat processor rates f_j for each task at time t. Thus, task A executes at speed $2/3$ and B at speed $1/3$ (see Figure 5.4).

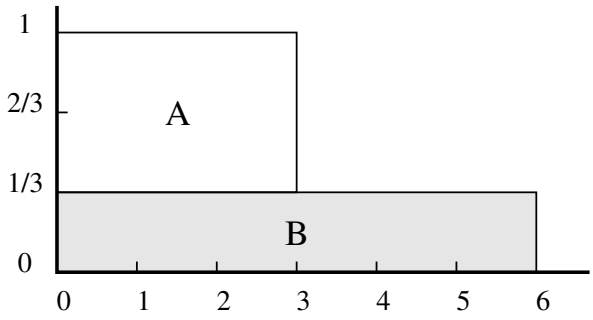


Figure 5.4. *Average Rate* schedule for case 1.

The *Average Rate* schedule is not optimal since A and B can be scheduled back to back as shown in Figure 5.5. Running at a constant speed is more energy efficient than with a varying speed due to the utility curve of the processor. The processor speed must be contant in any interval that contains no starting time and deadline time of a job.

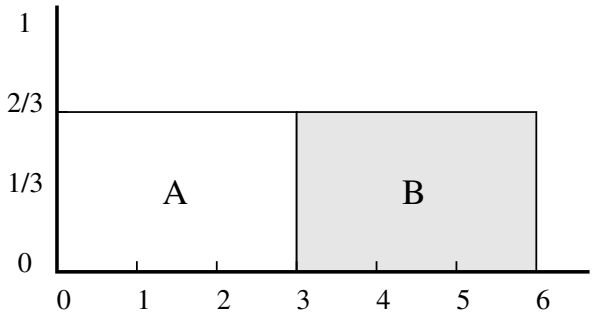


Figure 5.5. Optimal schedule for case 1.

A first improvement to the *Average Rate* heuristic is to take into account the other tasks already scheduled. When scheduling a next task, we can compute the (water) level above the current schedule (contour) to fit in the computational demands (area) of the task. The *task leveling* idea is outlined in Figure 5.6.

Applying task leveling to the first example yields the optimum (Figure 5.5) when task A is scheduled first, followed by B. However, scheduling B first and then A still yields the inferior schedule shown in Figure 5.4.

Our second improvement is to account for overlapping tasks that can be pushed aside. Consider the second case in Table 5.2, which adds a third task C to the optimal schedule in Figure 5.5. First note that task leveling fails to find a suitable schedule in this case since C must be layered on top of

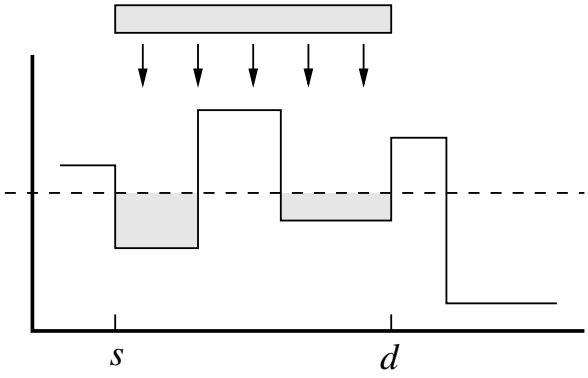


Figure 5.6. Task leveling.

B, raising the processor utilization above 1. The following method does find the optimal schedule (an equal load of $5/6$ across the entire $[0, 6]$ interval). In step one we determine the maximum processor utilization u_{max} on the interval $[s_C, d_C]$, which is $2/3$ (cf. interval $[4, 6]$ in Figure 5.5). In step two we fill up the free space below level u_{max} on interval $[s_C, d_C]$; this has no effect in our example because there is no space available. In the third step we determine all overlapping tasks (the set T) that overlap with C; T equals $\{\text{B}\}$. In the fourth step we compute the water level ($5/6$) above the contour of T+C that accommodates the remainder of C. Finally, we reschedule tasks from the set T to create space in the interval $[s_C, d_C]$; see Figure 5.7.

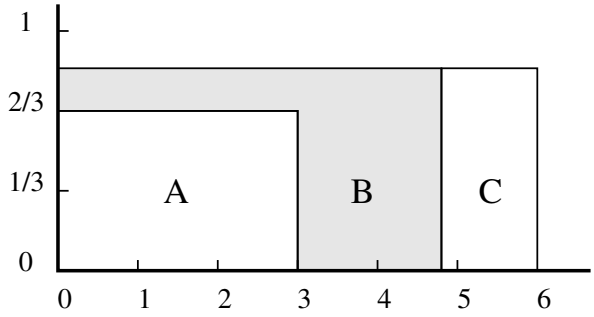


Figure 5.7. Optimal schedule for case 2.

Rescheduling in the final step is not always possible due to deadlines for tasks T, in which case steps four and five must be repeated. Dealing with overlapping tasks greatly enhances the quality of the clock schedules. Further improvements can be expected to also account for tasks that overlap with the overlapping tasks, etc. We do not pursue this direction, but rather arrange that tasks are scheduled in ascending priority. Tasks with relaxed deadlines (f_j close to 0) and few overlaps (low N_j) are ranked to be scheduled first, so they can easily make room for more difficult tasks if these come up for scheduling later on.

The details of our energy priority scheduler are presented in Algorithm 1. Step 2 calculates the priorities of the tasks. For example, in case 2 above, the priorities are set to $p_A = \frac{2}{3}\ (\frac{2}{3} \times 1)$, $p_B = \frac{5}{18}\ (\frac{2}{6} \times \frac{5}{6})$, and $p_C = \frac{1}{2}\ (\frac{1}{2} \times 1)$. Therefore EPS will schedule task B first, then C, and finally A. Note that this order is independent of the actual task arrivals, which avoids the sensitivity observed for the simpler heuristics discussed above. In steps 3.2.n a part of task j is scheduled by raising the "water" up to the next level. This level is to be found on the interval that includes all overlapping

Algorithm 1 Energy Priority Scheduling

0 Given a set of tasks T, each task with a starting time, deadline time, and fastest execution time.
1 Partition interval $[s_{min}, d_{max}]$ into a set of time regions $tr_i[start_i, end_i]$ where $start_i$ and end_i are start or deadline times of T, and there exists no other start or deadline time within tr_i.
2 For each task compute its priority: $p_j = f_j N_j$.
3 Repeat $\mid T \mid$ times:
3.1 Select task j that is not scheduled yet and has lowest p_j.
3.2 Repeat until task j is fully scheduled:
3.2.1 Determine intervals $tr_i \subseteq [s_j, d_j]$ with lowest scheduled processor utilization u_i
3.2.2 Determine overlapping task intervals tr_l, $tr_l \subseteq [s_t, d_t]$, util($t$,$tr_i$) > 0, $t \neq j$
3.2.3 Determine spill intervals $tr_k \in \{tr_l\} \setminus \{tr_i\}$, $u_k = u_i$
3.2.4 Define

$$
\begin{aligned}
u_{up} &= \text{lowest processor utilization on } \{tr_l\} \setminus \{tr_k\} \\
&\quad \text{(or 1 if } \{tr_l\} \setminus \{tr_k\} = \emptyset\text{)} \\
L_i &= \textstyle\sum \parallel tr_i \parallel \\
L_k &= \textstyle\sum \parallel tr_k \parallel \\
\delta &= \begin{cases} \min(u_{up} - u_i, \frac{\text{remain}(e_j)}{L_i}) & \text{if } L_k = 0 \\ \min(\frac{L_i u_i}{L_k}, \max(u_{up}, \frac{\text{remain}(e_j)}{L_i}) - u_i) & \text{otherwise} \end{cases}
\end{aligned}
$$

3.2.5 Set processor utilizations u_l to $u_i + \delta$ and reschedule tasks (including j) on tr_l and tr_i accordingly.
4 Regroup tasks spread across multiple intervals.

tasks. Time is divided into time regions, such time regions are used for detecting overlap. A new time region begins on any start and deadline time of a task. The spill intervals are the time regions of the overlapping tasks, not including $[s_j, d_j]$, where the processor utilization is equal to u_i and the utilization will be increased to make room for task j. Note that we only consider overlapping tasks that are actually scheduled on tr_i by including the 'util(t,tr_i) > 0' condition. If there are no spill intervals ($L_k = 0$), for example, when the first task is scheduled, the remaining work of j will be scheduled on top of the tr_i time regions. Otherwise, the work of the overlapping tasks is spilled from the tr_i intervals to the tr_k intervals. The actual increase (δ) is bound by the amount of work that can be spilled ($L_i u_i$), by the remainder of j that still needs to be scheduled, and the step up ($u_{up} - u_i$). One can efficiently implement the incremental scheduling of task j in steps 3.2.n by maintaining the overlapping intervals as a sorted list (ascending processor utilization). Once the final schedule is determined, tasks tend to be scattered over multiple intervals. To minimize the number of context switches, we regroup tasks in step 4 by swapping workloads between intervals.

The energy priority scheduling heuristic does not always find the optimal schedule, since it only accounts for pushing aside tasks that directly overlap with j. If non-overlapping tasks were also rescheduled in step 3.2.n, both the complexity and the ability of EPS to find the optimal schedule would increase. A heuristic such as EPS will fail to find the optimal schedule in complex workloads with many tasks. For example, when case 2 is slightly modified by changing task B to start at time 2, the insertion of task C will not raise the "water" above interval [0,2] as it could when realizing that B in turn should push task A aside. Fortunately, such workloads are not common for portable devices, where users typically run one or two concurrent applications. The complexity of the heuristic depends on the number of iterations needed to schedule j. In the worst case, each interval tr_i causes one step up. The maximum number of intervals is $2n-1$, leading to the upper bound of $O(n^3)$ for the complete

heuristic. In practice, one or two iterations often suffice and the number of overlapping tasks is small, lowering the complexity to $O(n \log n)$.

The presented energy priority algorithm makes a completely new schedule each time a new task arrives. When implementing this algorithm, several additions must be made such as properly updating the task list T when a new task arrives and the current running task is not yet finished. As an incremental version of the scheduling algorithm we use the following procedure: each time a new task j arrives, the set of intervals tr_i is extended, followed by one round of scheduling for task j (no looping over all tasks in step 3).

The energy priority algorithm must support sporadic tasks in a real-time OS context. The algorithm can support periodic tasks by adding a parameter w that indicates the window size for periodic task scheduling. Before step 0, every periodic tasks is converted in up to w sporadic tasks and added to the task set T. The periodic task with the shortest repetition period r_{min} bounds the interval $[0, r_{min}w]$ in which the periodic tasks are converted into multiple sporadic tasks. For example, T is extended with A ($r_a = 5, e_a = 2$) and B ($r_b = 9, e_b = 3$). When $w = 10$, the interval $[0, 50]$ is scheduled with 10 sporadic tasks A and 6 sporadic tasks B.

5.4 Results

To demonstrate the effectiveness of the Perfd framework we have performed power measurements on the LART platform (Section 3.3) consisting of variable-voltage hardware, an OS driver, a clock scheduling daemon and algorithm, and a power-aware video decoder.

5.4.1 Experimental platform

The LART has a programmable voltage regulator to control the voltage of the processor core. In Section 2.1.2 we already discussed the relation between processor frequency (59 - 236 MHz, steps of 14.7 MHz) and core voltage (0.79 - 1.5 V), see Figure 2.3.

The LART runs the Linux operating system (Version 2.4.0), which has been enhanced to support frequency and voltage scaling. We added a kernel module that reads the required frequency from `/proc`, a Linux pseudo filesystem used as a generic interface to kernel data structures, changes the clock frequency, and adjusts the core voltage. It subsequently recalibrates the kernel's internal delay routines, in particular those that busy-wait by counting instruction cycles. In addition, the kernel module adjusts the memory parameters that control the timings of the read/write cycles on the external bus. The code has been structured such that it may be interrupted and does not depend on external memory, which is temporarily unavailable during a clock frequency change. The SA-1100 is not designed for on-the-fly clock frequency changes. All DMA transfers are interrupted during a change, causing problems for DMA transfers of LCD video data. The LCD device driver needs to be informed of frequency changes and temporarily halt the DMA transfer. All LART design schematics and kernel modules are publically available [10].

We implemented a clock scheduler that mediates between applications and the basic OS driver controlling the core voltage and processor speed. To minimize implementation effort at the application level we designed the clock scheduler such that it supports both unmodified applications as well as power-aware applications specifying their future needs (AET and deadline). We use a combination of interval scheduling (for handling unknown workloads) and energy priority scheduling (supporting power-aware applications). We call the combined clock scheduler *PowerScale*. For convenience PowerScale is implemented as a daemon process in user space, but it can be moved inside the kernel

when the need arises. An application connects to PowerScale using a UNIX socket and specifies its workload as a set of tasks with starting times, deadlines, and processing needs (cycle count, minimum speed, or AET). Before running the energy priority scheduling (EPS) algorithm, PowerScale empties all sockets to consider at once all tasks currently made available by the power-aware applications. The computed schedule is then executed in a loop, listening on the socket for new tasks by invoking select() with a time-out value matching the time to the next speed change.

The interval-based component of PowerScale serves two purposes: it supports traditional applications and it corrects for mispredicted workloads by power-aware applications. Traditional applications do not register their workload with EPS, hence the speeds determined by EPS will be too low. Mispredicted workloads can cause EPS to determine a speed that is too low or too high. The interval scheduler within PowerScale monitors the Linux process scheduler statistics to see whether or not the EPS schedule needs to be adjusted. When the system load (processor utilization) is close to 1, the CPU is running at the right speed. Otherwise, the speed is adjusted: an overload (util $= 1$) is handled by increasing the speed, an underload (util < 0.5) is handled by reducing the speed. In effect the interval scheduler provides negative feedback to the speed schedule produced by EPS. To ensure stability the interval scheduler uses relatively long intervals in which EPS can issue multiple speed changes. Therefore the speed correction is applied as a delta (e.g., two steps up) to the rapidly changing EPS schedule.

The interval scheduler itself is quite flexible and operates with a parametrized interval length (a multiple of 10 ms, the granularity of Linux's 100 Hz internal timer). This allows it to operate stand alone (i.e. without EPS); a short interval length should be used to be able to closely follow the changes in the workload. To further improve the responsiveness of the system we employ the following heuristic. On consecutive speed increments we double the correction factor (exponential increase). On consecutive speed decrements, however, we simply step down to the next lower correction (linear decrease) since running the system at a speed that is too high does not impact its responsiveness; it only wastes energy. The correction factor (delta) is applied to a fixed maximum performance schedule (236 MHz).

A modification of the power-aware video decoder was required to work around the poor granularity of the internal Linux timer (100 Hz). The H.263 decoder has a simple rate control mechanism for displaying the frames at the specified rate (15 fps): after decoding a frame it computes the time left until the next display deadline, and invokes the `usleep()` system call to wait for that time to pass before outputting the video frame. `Usleep()` may return up to 10 ms late due to the poor Linux timer granularity, which is a significant part of the frame time (67 ms). Each delay causes a frame deadline miss, and must be compensated for in the next frame to catch up. When running at a constant high speed, the H.263 decoder does this automatically by waiting a bit shorter in the next frame. When scaling speeds, however, we must explicitly account for the inaccuracy by overestimating the computational demand of each frame. We took a drastic approach and replaced the `usleep()` call with a busy-wait loop, in which we read the clock until the next display deadline is met.

5.4.2 Video decoder modes

We used the experimental setup discussed in Section 5.4.1 to measure the power consumption of our extended H.263 decoder (Section III) on top of PowerScale.

Table 5.3 shows the average power consumption of the LART platform for decoding a test sequence for the three supported modes of the decoder: feed backward (FB), feed forward (FF), and optimal (opt). For comparison the "236" column shows the average power consumption with clock scheduling disabled and using a fixed clock frequency of 236 MHz. The average power is computed

Sequence	total (core + fixed) 236	FB	FF	opt	fixed
Grandma	404.6	244.5	243.1	242.2	209.6
Salesman	417.0	262.7	254.2	245.5	208.3
Trevor	496.1	362.8	355.6	347.9	249.4
Carphone	556.5	368.7	357.4	351.2	263.5
Foreman	571.6	389.7	380.3	374.7	283.5

Table 5.3. Average power consumption [mW].

by measuring the total energy consumed by the LART and dividing that by the duration of the test sequence. The test sequences are stored in the RAM-disk provided by Linux, hence, little energy is needed to retrieve them.

The measurements show that the FB mode reduces energy consumption considerably compared to running at 236 MHz, for example: the average power dissipated when decoding the Grandma sequence drops from 404.6 mW to 244.5 mW. The reduction for FB ranges from a factor of 1.37 (Trevor) to a factor of 1.66 (Grandma). Providing the decoder with additional information (FF and optimal) does indeed reduce energy consumption further, but the gain is limited. In the case of FF the reduction ranges from 1.40 (Trevor) to 1.67 (Grandma). The optimal policy achieves reductions in the range of 1.43 (Trevor) to 1.70 (Salesman).

The differences between the various policies are small because voltage scaling only reduces the power consumption of the processor core. The last column in Table 5.3 presents the power consumed by the components (memory, bus, etc.) supplied from the fixed 3.3 V. It shows that the fraction of the total power that can be attributed to non-CPU subsystems is considerable. For example, when decoding the Grandma sequence at 236 MHz, 209.6 mW out of 404.6 mW are consumed by non-CPU subsystems. The fixed fraction of the total power consumption ranges from 47.4 % (Carphone) to 51.8 % (Grandma). As a consequence, the maximum power reduction that can be obtained by controlling the processor speed and core voltage is limited to roughly a factor of two. We expect, however, that this limit can be increased by optimizing the H.263 decoder to take into account the size of the cache, which is part of the scalable processor core, to reduce the memory traffic. For example, large look-up tables are ineffective on the LART with its small data cache of 8 KB, and degrade performance.

When considering only the power consumed by the processor core, FB achieves a significant reduction of 2.25 (Trevor) to 6.45 (Grandma). The reduction by FF ranges from 2.33 (Trevor) to 6.63 (Grandma), and the optimal policy results in a reduction of 2.59 (Trevor) to 7.02 (Salesman). The relatively small difference between the FB, FF, and optimal mode indicates that a priori knowledge on frame length (FF) or complete processing requirements (opt) provides only a small benefit. Thus, standard H.263 video sequences can be decoded efficiently (power wise) using the feedback mode.

5.4.3 Cooperative versus interval scheduling

To show the advantage of cooperative clock scheduling over interval scheduling, we study the behavior of the bursty video-decoding application in detail. We use the Carphone test sequence since subsequent frames in this video often differ considerably in size and sometimes in type (see Figure 5.2). Decoding a frame at a speed that is too high wastes energy; using one that is too low results in a missed deadline. Our modified H.263 decoder reports the accumulated miss-times at the end. With this quality measure it is possible to study the trade-off between power and quality. Our accumulated

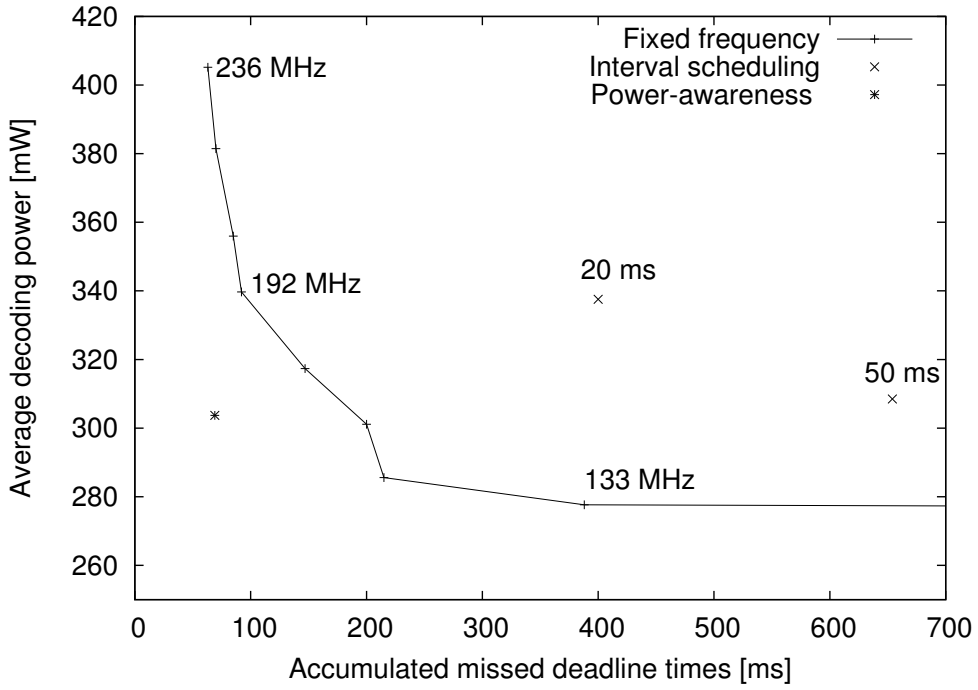


Figure 5.8. Power-quality tradeoff.

miss-times metric is similar to the clipped-delay metric in the simulations by Pering [145].

Figure 5.8 shows the power-quality trade-off for cooperative clock scheduling, interval scheduling, and decoding at fixed speeds. Note that in all cases deadlines are missed. This is caused by the initial I-frame in the Carphone sequence that cannot be decoded within 67 ms, even at the highest frequency. The solid line in Figure 5.8 shows the effect of decreasing the (fixed) frequency from 236 MHz (405 mW, 63 ms) down to 133 MHz (278 mW, 388 ms). The power consumption goes down at the expense of additional deadline misses since the number of frames that cannot be decoded within 67 ms increases when the clock frequency lowers.

In interval-based mode PowerScale performs worse than running at a fixed speed. For example, with a 20 ms interval setting PowerScale operates with an average power of 337 mW and causes 400 ms of missed deadlines; running at a fixed speed of 192 MHz requires the same power, but reduces the missed deadlines to only 92 ms, while running at 133 MHz incurs a similar miss time, but requires less power (278 mW). The problem for the interval scheduler is that a short-time average is not a good predictor of the speed at which to decode the next frame. Increasing the interval length makes the scheduler behave more like a fixed-speed scheduler; with a 50 ms interval the power consumption gap to the fixed schedules (solid line) is smaller than at 20 ms, but many more deadlines are missed. Without additional knowledge an interval scheduler will never be able to handle bursty workloads well.

Using the AET information from the power-aware video decoder results in substantial power savings since the workload description allows PowerScale (EPS mode) to select the right decoding speed in most cases. The decoding of the Carphone sequence requires only 304 mW (100 mW CPU, 204 mW non-CPU), and misses just a few deadlines: 67 ms in total, of which the largest part is caused by the too-demanding initial I-frame. For comparison, decoding at the fixed frequency of 236 MHz consumes 405 mW (198 mW CPU, 207 mW non-CPU) and delivers the same quality: 63 ms of

accumulated deadline misses. Thus, cooperative voltage scaling reduces the power consumption of the processor core with a factor of two. The total system power, however, is only reduced by 25 % because of the power consumed by the non-CPU subsystems supplied by the fixed 3.3 V.

5.4.4 Multiple applications

We now demonstrate the ability of the EPS algorithm to combine the processing needs of multiple applications and create a power-efficient clock schedule. In the following experiment the Carphone sequence is decoded in conjunction with a synthetic application. Both the video decoder and the synthetic task register their processing requirements (AETs and cycle count, respectively) with the PowerScale scheduler. We log the speed changes initiated by PowerScale during the experiment, and measure the power consumption of the processor core. The solid line in Figure 5.9 shows the actions of the PowerScale scheduler for one second of the benchmark video (frames 135-150). The synthetic application places a reservation for 150 ms of processor time at a speed of at least 40 MHz during the same time at which frames 141-143 must be decoded. The curve shows how the processor speed changes over time in our experiment (each frame takes 67 ms). The shaded area shows the impact of the synthetic task on the EPS schedule: the speed is raised to 207 MHz. For comparison the dotted line in Figure 5.9 shows the behavior of PowerScale when running in interval-based mode. The resulting speed is either too low (e.g., frame 136) or too high (e.g, frames 144-150).

We carefully crafted the combined workload to contain overlapping tasks. The synthetic task enters the system 25 ms after frame 141 starts and must finish 25 ms before frame 143 ends; the start-stop interval is indicated in Figure 5.9. The synthetic task thus overlaps with frames 141, 142, and 143. The EPS algorithm schedules the synthetic task first, because it has the lowest flat processor rate (40 MHz), followed by frame 141 (148 MHz), 142 (162 MHz), and 143 (207 MHz). The final schedule raises the processor speed during the decoding of frames 141 and 142 (i.e. the shaded area in Figure 5.9). This effectively creates a 30 ms gap between frame 141 and 142, which contains enough cycles to run the synthetic task ($30 \times 207 > 150 \times 40$).

The measured power dissipation of the processor (Figure 5.10) shows a shape that is quite similar to the clock schedule executed by PowerScale in EPS-mode (Figure 5.9). Note, however, that the peak-to-bottom power ratio is larger than the corresponding speed ratio. Neglecting frame 137, which requires no computation and causes the processor to enter its special idle mode, the peak-to-bottom power ratio is around 6 ($\text{frame 136} : \text{frame 135} \approx 271 : 43$), the speed ratio is around 2.5 ($221 : 89$). This shows the effect of the quadratic relation between power and voltage. The exact time location of the synthetic task is marked in Figure 5.10. Its power consumption is larger than that of its neighboring decoding tasks running at the same speed because the synthetic task does not reference main memory, hence, incurs no processor stalls when waiting for memory accesses to complete.

Figure 5.10 shows the power dissipated by the CPU only. Figure 5.11 shows the power dissipation of the 3.3 V part of our system (memory, bus, etc.). Despite the clock speed changes induced by PowerScale, the system load in Figure 5.11 shows a quite regular pattern, except for frames 136, 137, and 142. The decoding of frames 136 and 137 involves a PB sequence where all computation is performed in the first frame (136), hence the "zero" power consumption in frame 137. The drop to zero in frame 142 is caused by the execution of the synthetic task that does not reference any memory, but only exercises the CPU. The high peaks at the beginning of each frame are caused by the video decoder performing the run-length decoding of the compressed frame. This involves fetching data from the RAM disk, where the Carphone sequence is stored, into the cache over the external bus. After this burst of memory traffic the decoder starts processing the data, which requires more computation, and the power dissipation of the memory subsystem drops to a level around 200 mW.

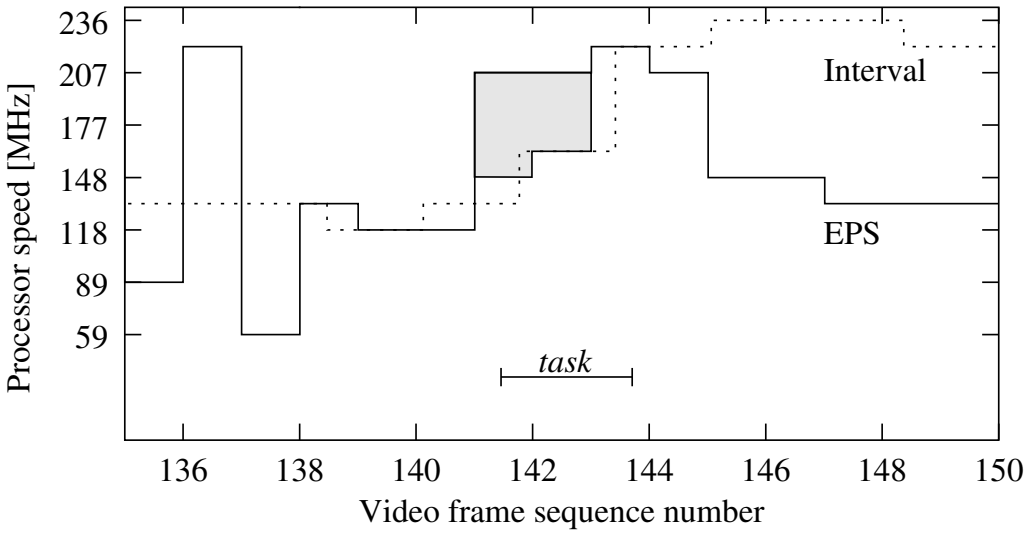


Figure 5.9. Clock schedules executed by PowerScale.

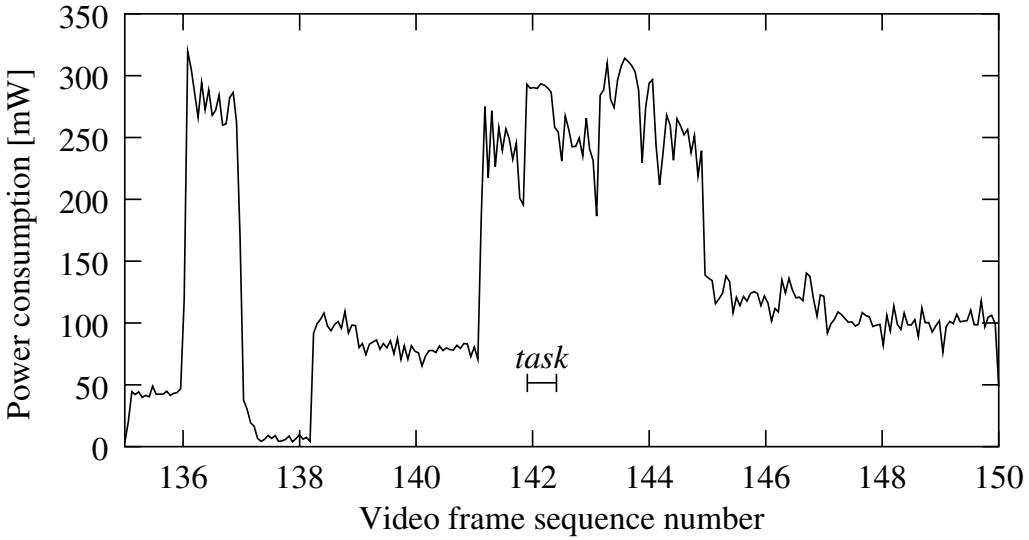


Figure 5.10. Processor power consumption of EPS.

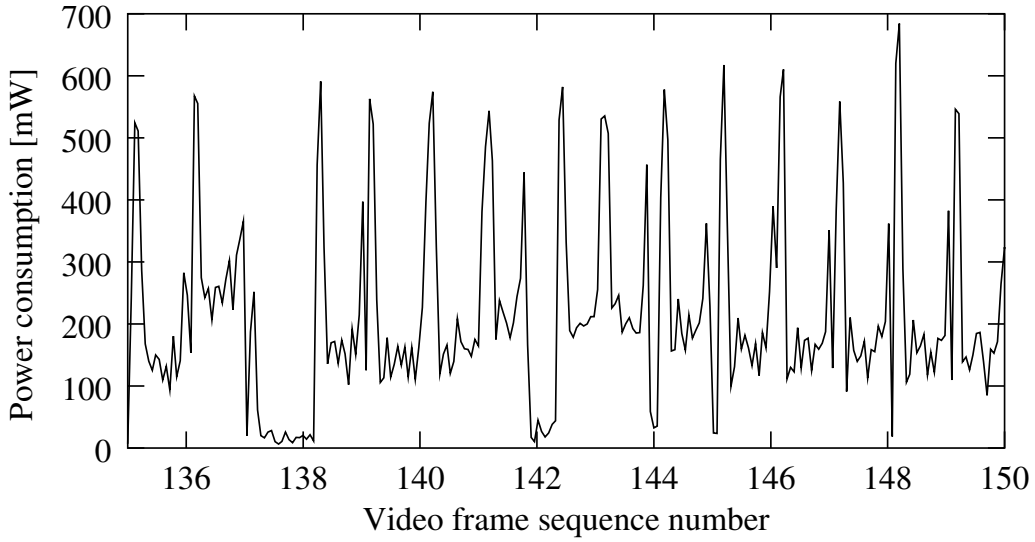


Figure 5.11. Non-CPU power consumption of EPS.

Note that the average non-CPU power (202 mW) exceeds the average processor power (118 mW), which limits the overall effectiveness of voltage scaling.

5.5 Conclusions

As stated in Section 3.1.1, policies currently in use on portable devices are simple. A sophisticated policy is required to efficienctly control a component with an advanced mechanism such as the PowerPC 405LP (Section 3.2.4). The 405LP requires a policy with a mechanism control resolution (Section 2.3.3) of "full setting". As of 2003, this sophistication is beyond the state-of-the-art of policies integrated in operating system such as Linux 2.5 and Windows XP. The Linux "cpufreq" project, which we seeded in 1999 by writing the first Linux OS device driver for voltage scaling, still lacks an intelligent policy. Policies have only recently matured enough to produce the "performance level" of mechanism control resolution shown in Figure 3.9. Because of the Perfd framework, the architecture is advanced enough to move to the next level and exploit mechanisms with "full setting" mechanism control resolution.

Our implementation of the Perfd processing component with our EPS scheduling algorithm is the first to exploit cooperation to this level. To demonstrate the effectiveness of application-directed clock scheduling we have actually build a complete system consisting of variable-voltage hardware (LART), OS support (Linux driver), intelligent policy (PowerScale), clock scheduling algorithm (EPS), and a cooperative application (H.263 video decoder).

We measured and analyzed the effectiveness of cooperative clock scheduling with a workload consisting of a Perfd-aware video decoder competing with a computational task. The results show that EPS successfully schedules both applications and reduces the energy consumption of the processor with 50% when compared to running at full speed (236 MHz). This is a significant improvement over interval-scheduling achieving 33% reduction. EPS achieves its reduction without missing deadlines, unlike interval scheduling that does miss deadlines. The processor only consumes a portion of the total system power. When compared to running at full speed, EPS reduces the system power with

25 %.

The power efficiency improves significantly when applications provide a small amount of information (AET estimation) to the cooperative clock scheduler. With these experiment we have successfully demonstrated the ability of the Perfd framework to improve power efficiency through cooperation.

Chapter 6

Conclusions

THIS is the final chapter of this thesis. This chapter reflects on all previous work and provides concluding remarks at a higher level of abstraction than the remarks at the end of each chapter. This chapter also includes a section that presents possible enhancements and extensions of the presented research.

6.1 Summary

This thesis deals with power management on portable devices and views it from both a theoretical perspective (Chapters 1, 2, and 3) and a practical perspective (Chapters 4 and 5). We have given an introduction to portable devices and to the trends in their technological development (Chapter 1).

We have defined the "Delft taxonomy" of power management that provides the first structured listing of all known approaches to power management (Chapter 2). The Delft taxonomy provides insight into the scientific progress in the field of power management. Our taxonomy clearly shows the increasing sophistication of power-management solutions in the last decade (1993-2003). In 1993, power management was still limited to only exploiting sleep modes, using a power management policy based on a fixed idle timeout [45]. Recently, more sophisticated policies have been developed that adapt based on changed resource cost and benefit [197] and policies that adapt the performance for battery lifetime control [199].

Despite the many advances in sophistication of power management policies, one fundamental problem remained unsolved. The various power management policies within a portable device operate in isolation. The model of interaction between the components of a portable device has not been improved in the last decade. Power management policies do not cooperate. It has been argued in [16; 17; 55; 139] that cooperation has potential in the area of power management, but that no solution exists. We addressed this issue and developed a solution called the "Perfd framework" (Chapter 3).

The Perfd framework provides an infrastructure for cooperative power management. Cooperating policies are based on two foundations: context awareness and sharing of non-functional information such as power consumption, performance metrics, future resource requirements, end-user benefit estimation, and configuration alternatives. The concept of context awareness means having the sophistication to understand and interact with one's environment and react to changes in that environment. When policies share more information, uncertainty is lowered and future events can be anticipated on. The access to information improves the efficiency to what level a policy can exploit a power management mechanism. We have identified the access to information as a key parameter that bounds the efficiency of a policy. We have called the level of information access for a policy "input richness".

In this thesis, we argue that significant improvements in power management are only possible when more information is made available to policies. Our experiments have proven this claim.

Context awareness of a policy implies the sophistication to make trade-offs. We have demonstrated the ability of our Perfd implementation to make trade-offs at execution time (Chapter 4). Our software was able to determine the cost in terms of power consumption for several alternative courses of action. Our implementation was able to compile a list of alternatives and select the most power-efficient course of action at the cost of only a small amount of CPU time. (1.1 ms in our experiment).

We successfully lowered the power consumption of an embedded computer running a video decoder. We developed a cooperating policy to exploit the voltage scaling mechanism of the processing component. By maximizing the input richness for the cooperating policy and development of a new scheduling heuristic (Energy Priority Scheduling), we were able to reduce the power consumption of the processor core with a factor of two (Chapter 5).

6.2 Reflections

The Delft taxonomy enabled us to see an emerging pattern in the development of power management solutions. The Delft taxonomy and the identification of input richness provided the insight that cooperation was an unexplored opportunity with significant potential. We have developed the Perfd framework specifically to exploit this opportunity. Our Perfd framework clearly fits the general direction of power management solutions with ever increasing sophistication.

Our experimental validation of the Perfd framework (Chapters 4 and 5) was not an isolated activity. The Perfd framework itself was improved and formed by both careful design and feedback from experiments during five years of research. We therefore claim that the Perfd framework is not merely an idea, but the first realization of the cooperative power management concept; proven on actual hardware with realistic applications. This approach radically differs from the approach taken in many publications that divide the research into two isolated stages. First is the development of a power management approach. Second is the evaluation using simulations.

In our view the ultimate goal in mobile computing is “making portable devices more useful and intelligent” (Chapter 3). We will now reflect on the contribution of the Perfd framework to the usefulness and intelligence of portable devices.

We define usefulness as the ability to support the user in his or her daily life (private and professional). Battery lifetime is an important factor in the usefulness and this thesis therefore provides a direct contribution to this goal. The ability to predict the resource cost of services using Perfd provides the opportunity to accurately estimate the remaining battery lifetime. Furthermore, due to cost prediction and support for reacting to changing conditions in Perfd, it is even possible to offer a new functionality we call “guaranteed battery lifetime”.

The more difficult question is whether this thesis contributes to the quest for intelligent portable devices. Intelligence is very difficult to define [159]. Most definitions include the following elements: logical reasoning, planning of actions, and ability to understand and manipulate the environment. By enabling cooperative power management, the Perfd framework enables more intelligence. The exchange of non-functional information, reasoning on config-space graphs, cost prediction using performance models, and the ability for pro-active behavior all contribute to more intelligence.

The gains of cooperative power management comes at a cost. Each OS device driver needs to be modified, applications need to be enhanced, and Perfd drivers must be created. Such Perfd drivers consist of a scheduler, a configurator, and a performance model. We were able to program two Perfd drivers in two weeks, showing that the required programming effort is relatively small.

The ultimate proof for the success of the Perfd framework is the level of acceptance by the research community and software developers. The software developers (a.k.a The Market) will determine if the costs of using Perfd in terms of a modest increase of software size, complexity, and execution-time overhead are worth the significant gains in usefulness and intelligence.

6.3 Further extensions

We have developed the first framework for cooperative power management and implemented a prototype. Research, however, is never finished and a thesis can always be expanded. For example, by conducting more experiments and improving the functionality.

In this thesis we have conducted experiments with two hardware platforms and several software components. These experiments can be extended with other hardware platforms such as mobile phones and more software components. The mobile phone platform is particularly suited for the Perfd framework. Power management is still very static in this field. For example, a mobile GSM phone contacts the network every few seconds and checks for incoming calls. By varying the incoming-call check-interval, the standby time of a phone can be increased. Such features strictly do not require Perfd, but are much easier to implement with a framework for cooperation.

Development of cooperating software components is another interesting extension of our research. A great need exists for a generic audio/video handling component with plug-ins for all available formats with an open source software license. Existing audio/video handling components are not resource aware, do not allow extensive trade-offs, and have no built-in performance model. Another opportunity is in the area of wireless links. The 802.11 standard includes many obstacles and opportunities for power management. Improvements in input richness, enabled by Perfd, have significant potential. For example, the method used in 802.11 for checking the network for incoming data packets consumes significant power. A special signalling channel with a lower data rate would be much more efficient. Perfd provides the infrastructure for a policy that control this signalling channel.

This thesis presented experiments that exploited the novel idea of using performance models during execution time. However, using performance models in power management is still not fully understood. The optimal accuracy problem remains unsolved (Section 3.2.3). More research and experimentation is needed to determine what the optimal accuracy of a performance model should be for several scenario's and how many trade-offs of the hardware must be included in the model.

Bibliography

[1] S. Ahuja, N. Carreiro, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.

[2] O. Angin, A. Campbell, M. Kounavis, and R. Liao. The MobiWare toolkit: Programmable support for adaptive mobile networking. *IEEE Personal Communications*, August 1998.

[3] Semiconductor Industry Association. The international technology roadmap for semiconductors 2001 edition. December 2001.

[4] Atheros-corporation. The ar5001x combo wlan solution. `http://www.atheros.com/pt/ComboBulletin.pdf`.

[5] C. Aurrecoechea, A. Campbell, and L. Hauw. A survey of QoS architectures. *Multimedia Systems*, 6(3):138–151, 1998.

[6] A. Azevedo, I. Issenin, R. Cornea, R. Gupta, N. Dutt, A. Veidenbaum, and A. Nicolau. Profile-based dynamic voltage scheduling using program checkpoints. In *Design Automation and Test in Europe DATE*, pages 168–176, March 2002.

[7] B. Badrinath, A. Fox, L. Kleinrock, G. Popek, P. Reiher, and M. Satyanarayanan. A conceptual framework for network and client adaptation. *IEEE Mobile Networks and Applications*, 5(4), 2000.

[8] J.-D. Bakker. LART – design of a low-power embedded Linux machine. *Journal of Linux Technology*, 1(3):8–19, 2000.

[9] J.-D. Bakker, K. Langendoen, and H. Sips. LART: Flexible, low-power building blocks for wearable computers. In *Int. Workshop on Smart Appliances and Wearable Computing (IWSAWC)*, Scottsdale, AZ, April 2001.

[10] J.-D. Bakker, J.A.K. Mouw, and M.A.H.G. Joosen. Linux Advanced Radio Terminal design page. `http://www.lart.tudelft.nl/`.

[11] J.-D. Bakker and F. Schoute. LART: design and implementation of an experimental wireless platform. In *Proceedings of the 21st Symposium on Information Theory in the Benelux*, May 2000.

[12] H. Balakrishnan, V.N. Padmanabhan, S. Seshan, and R.H. Katz. A comparison of mechanisms for improving TCP performance over wireless links. *IEEE/ACM Transactions on Networking*, 5(6):756–769, 1997.

[13] S. Balakrishnan and J. Ramanan. Power-aware operating systems using ACPI. Technical report, University of Wisconsin, Computer Science Department, 2001.

[14] K.C. Barr. Energy aware lossless data compression. Master's thesis, Massachusetts Institute of Technology, September 2002.

[15] A.C. Bavier, A. Brady Montz, and L.L. Peterson. Predicting MPEG execution times. In *Measurement and Modeling of Computer Systems (SIGMETRICS)*, pages 131–140, June 1998.

[16] F. Bellosa. The benefits of event-driven energy accounting in power-sensitive systems. In *Proceedings of 9th ACM SIGOPS European Workshop*, Kolding, Denmark, September 2000.

[17] L. Benini and G. de Micheli. System-level power optimization: Techniques and tools. In *International Symposium on Low Power Electronics and Design*, 1999.

[18] Berger and Gibson. Lossy source coding. *IEEE Transactions on Information Theory*, 44, 1998.

[19] Y. Bernet. The complementary roles of RSVP and differentiated services in the full-service QoS network. *IEEE Computer Magazine*, 32(2):153–162, February 2000.

[20] A.J. Bernstein and J.C. Sharp. A Policy-Driven Scheduler for a Time Sharing System. *Communications of the ACM*, 14(2):74–78, February 1971.

[21] L. Benini L. Bogliolo and G. De Michelli. A survey of design techniques for system-level dynamic power management. *IEEE transactions on very large scale integration (VLSI) systems*, 8(3):299–316, June 2000.

[22] P. Bosch and S.J. Mullender. Real-time disk scheduling in a mixed-media file system. In *IEEE Real Time Technology and Applications Symposium*, 2000.

[23] B. Brock. An embedded-linux power-management framework. `http://www.linux.org.uk/mailman/private/cpufreq/2002-August/000758.html%`.

[24] N. Brown and C. Kindel. Distributed component object model protocol – dcom/1.0, 1996. Microsoft Corporation.

[25] L.O. Burchard and P. Altenbernd. Worst-case execution times analysis of MPEG-decoding. In *10th Euromicro Conference on Real Time Systems (WRTS)*, 1999.

[26] T. Burd and R. Brodersen. Processor design for portable systems. *Journal of VLSI Signal Processing*, August 1996.

[27] T. Burd, T. Pering, A. Stratakos, and R. Brodersen. A dynamic voltage scaled microprocessor system. In *IEEE International Solid-State Circuits Conference*, pages 294–295, February 2000.

[28] Thomas T. Burd and Robert W. Brodersen. *Energy efficient microprocessor design*. Kluwer Academic Publishers, Boston, US, 2002.

[29] A. Burns. Scheduling hard real-time systems: A review. *Software Engineering Journal*, 6(3):116–128, 1991.

[30] L. Buttyán and J.-P. Hubaux. Nuglets: a virtual currency to stimulate cooperation in self-organized mobile ad hoc networks. Technical Report DSC/2001, 2001.

[31] N. Carriero and D. Gelernter. The s/net's linda kernel. *ACM Transactions on Computer Systems (TOCS)*, 4(2):110–129, 1986.

[32] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[33] A. Chandrakasan, V. Gutnik, and T. Xanthopoulos. Data driven signal processing: an approach for energy efficient computing. In *International Symposium on Low Power Electronics and Design*, pages 374–352, August 1996.

[34] A. Chandrakasan, S. Sheng, and R. Brodersen. Low-power CMOS digital design. *IEEE Journal of Solid-State Circuits*, 27(4):473–484, April 1992.

[35] N. Chang, K. Kim, and H.-G. Lee. Cycle-accurate energy consumption measurement and analysis: Case study of ARM7TDMI. *IEEE transactions on very large scale integration (VLSI) systems*, 10(2), April 2002.

[36] R.Y. Chen, M.J. Irwin, and R.S. Bajwa. Architecture-level power estimation and design experiments. In *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, volume 6, pages 50–66, 2001.

[37] T.-W. Chen, P. Krzyzanowski, M.R. Lyu, C.J. Sreenan, and J.A. Trotter. Renegotiable quality of service - a new scheme for fault tolerance in wireless networks. In *Symposium on Fault-Tolerant Computing*, pages 21–30, 1997.

[38] H.-H. Chu and K. Nahrstedt. A soft real time scheduling server in UNIX operating system. In *Interactive Distributed Multimedia Systems and Telecommunication Services*, pages 153–162, 1997.

[39] E. Chung, Y. Huang, S. Yajnik, D. Liang, J.C. Shih, C.-Y. Wang, and Y.-M. Wang. DCOM and CORBA side by side, step by step and layer by layer. `http://www.cs.wustl.edu/~schmidt/submit/Paper.html`, November 1997.

[40] T.L. Cignetti, K. Komarov, and C. Ellis. Energy estimation tools for the palm. In *ACM MSWiM 2000: Modeling, Analysis and Simulation of Wireless and Mobile Systems*, August 2000.

[41] D.D. Clark, S. Shenker, and L. Zhang. Supporting real-time applications in an integrated services packet network: Architecture and mechanism. In *SIGCOMM*, pages 14–26, 1992.

[42] P. Collet. On contract monitoring for the verification of component-based systems. In *OOPSLA Workshop on Specification and Verification of Component-Based Systems (OOPSLA'2001)*, October 2001.

[43] Dallas Semiconductor Corp. Smart battery monitor (ds2437), July 2000.

[44] Digital Equipment Corporation, Hewlett-Packard Company, HyperDesk Corporation, NCR Coporation, Inc. Object Design, and SunSoft, Inc. The Common Object Request Broker: Architecture and specification. Technical Report 91-12-1, Object Management Group, Framingham MA (USA), December 1991.

[45] Intel Corporation and Microsoft Corporation. Advanced Power Management (APM) Specification Version 1.0, September 1993.

[46] K. Czajkowski, I.T. Foster, and C. Kesselman. Resource co-allocation in computational grids. In *HPDC*, 1999.

[47] E.F. Deprettere. Ubiquitous Communications, Program Proposal. Delft University of Technology, March 1997.

[48] P. Devanbu. The ultimate reuse nightmare: Honey, i got the wrong dll. In *5th Symposium on Sofware Reuseability*, pages 178–180, May 1999.

[49] R.P. Dick, G. Lakshminarayana, A. Raghunathan, and N.K. Jha. Power analysis of embedded operating systems. In *Design Automation Conference*, pages 312–315, 2000.

[50] H. van Dijk, K. Langendoen, and H. Sips. ARC: a bottom-up approach to negotiated QoS. In *3rd IEEE Workshop on Mobile Computing Systems and Applications (WMCSA 2000)*, pages 128–137, Monterey, CA, December 2000.

[51] F. Douglis, P. Krishnan, and B. Bershad. Adaptive disk spin-down policies for mobile computers. In *2nd USENIX symposium on Mobile and Location Independent Computing*, April 1995.

[52] F. Douglis, P. Krishnan, and B. Marsh. Thwarting the power-hungry disk. In *USENIX Winter*, pages 292–306, 1994.

[53] M. Doyle, T.F. Fuller, and J. newman. modeling of galvanostatic charge and discharge of the lithium/polymer/insertion cell. *Journal of the Electrochemical Society*, 140(6), June 1993.

[54] J.-P. Ebert, B. Stremmel, E. Wiederhold, and A. Wolisz. An energy-efficient power control approach for wlans. *Journal of Communications and Networks (JCN)*, 2(3):197–206, September 2000.

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

[56] C. Ellis, A. Lebeck, and L. Vahdat. System support for energy management in mobile and embedded workloads: A white paper. `http://citeseer.nj.nec.com/ellis99system.html`, October 1999.

[57] R. Ernst and W. Ye. Embedded program timing analysis based on path clustering and architecture classification. In *International Conference on Computer Aided Design*, pages 598–604, November 1997.

[58] L.M. Feeney and M. Nilsson. Investigating the energy consumption of a wireless network interface in an ad hoc networking environment. In *IEEE INFOCOM*, 2001.

[59] K. Flautner, S. Reinhardt, and T. Mudge. Automatic performance-setting for dynamic voltage scaling. In *7th ACM Int. Conf. on Mobile Computing and Networking (Mobicom)*, Rome, Italy, July 2001.

[60] J. Flinn and M. Satyanarayanan. Powerscope: a tool for profiling the energy usage of mobile applications. In *Second IEEE Workshop on Mobile Computing Systems and Applications*, pages 2–10, February 1999.

[61] I. Foster. The anatomy of the Grid: Enabling scalable virtual organizations. *Lecture Notes in Computer Science*, 2150, 2001.

[62] T.F. Fuller, M. Doyle, and J. Newman. Relaxation phenomena in lithium-ion-insertion cells. *Journal of the Electrochemical Society*, 141(4), April 1994.

[63] L. Geppert and T.S. Perry. Transmeta's magic show. *IEEE Spectrum*, 37(5), May 2000.

[64] R.A. Golding, P. Bosch II, C. Staelin, T. Sullivan, and J. Wilkes. Idleness is not sloth. In *USENIX Winter*, pages 201–212, 1995.

[65] J. Gomez, A. Campbell, M. Naghshineh, and C. Bisdikian. Power-aware routing in wireless packet networks. In *Sixth IEEE International Workshop on Mobile Multimedia Communications*, San Diego, CA, US, November 1999.

[66] K. Govil, E. Chan, and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *MobiCom*, Berkeley, CA, November 1995.

[67] F. Gruian. *Energy-centric scheduling for Real-Time Systems*. PhD thesis, Lund University, Sweden, December 2002.

[68] D. Grunwald, P. Levis, K. Farkas, C. Morrey, and M. Neufeld. Policies for dynamic clock scheduling. In *Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, CA, October 2000.

[69] S.H. Gunther, F. Binns, D.M. Carmean, and J.C. Hall. Managing the impact of increasing microprocessor power consumption. *Intel Technology Journal*, 2001.

[70] S. Gurumurthi, A. Sivasubramaniam, M.J. Irwin, N. Vijaykrishnan, M.T. Kandemir, T. Li, and L.K. John. Using complete machine simulation for software power estimation: The softwatt approach. In *Eighth International Symposium on High-Performance Computer Architecture (HPCA'02)*, February 2002.

[71] T. Dohi H. Okamura and S. Osaki. Performance analysis of a transaction based software system with shutdown. In *Second International Workshop on Software and Performance WOSP2000*, September 2000.

[72] H. Haertig, L. Reuther, J. Wolter, M. Borriss, and T. Paul. Cooperating resource managers. In *Fifth IEEE Real-Time Technology and Applications Symposium (RTAS)*, June 1999.

[73] A. Hafid, G. von Bochmann, and R. Dssouli. A quality of service negotiation approach with future reservations (NAFUR): a detailed study. *Computer Networks and ISDN Systems*, 30(8):777–794, 1998.

[74] W.R. Hamburgen, D.A. Wallach, M.A. Viredaz, L.S. Brakmo, C.A. Waldspurger, J.F. Bartlett, T. Mann, and K.I. Farkas. Itsy: Stretching the bounds of mobile computing. *IEEE Computer*, 34(4):28–37, April 2001.

[75] Handspring. Handspring treo developer technical information. `http://www.handspring.com/developers/treo_info_center.jhtml`, 2002.

[76] G. Hardin. The tragedy of the commons. *Science*, 162:1243–1248, 1968.

[77] M.G. Harmon, T.P. Baker, and D.B. Whalley. A retargetable technique for predicting execution time. In *IEEE Real-Time Systems Symposium*, pages 68–77, 1992.

[78] E. Harris, S. Depp, W. Pence, S. Kirkpatrick, M. Sri-Jayantha, , and R. Troutman. Technology directions for portable computers. *Proceedings of IEEE*, 83(4), April 1995.

[79] P.J.M. Havinga. *Mobile Multimedia Systems*. PhD thesis, University of Twente, February 2000.

[80] D.P. Helmbold, D.D.E. Long, and B. Sherrod. A dynamic disk spin-down technique for mobile computing. In *Second ACM conference on Mobile Computing and Networking (MobiCom)*, pages 130–142, 1996.

[81] G.J. Henry. The Fair Share Scheduler. *AT&T Bell Laboratories Technical Journal*, 63(8):1845–1857, October 1984.

[82] J. Hillston. *A Compositional Approach to Performance Modelling*. PhD thesis, University of Edinburgh, April 1994.

[83] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. Srivastava. Power optimization of variable voltage core-based systems. In *36th Design Automation Conference*, pages 176–181, June 1998.

[84] C. Hsu, U. Kremer, and M. Hsiao. Compiler-directed dynamic frequency and voltage scaling. In *Workshop on Power-Aware Computer Systems*, 2000.

[85] C.-H. Hwang and A.C. Wu. A predictive system shutdown method for energy saving of event-driven computation. In *Int. Conf. Computer-Aided Design*, pages 28–32, November 1997.

[86] IBM. Adaptive power management for mobile hard drives. White Paper, Jan 1999.

[87] T. Ikeda. Thinkpad low-power evolution. *IEEE Symposium on Low Power Electronics*, October 1995.

[88] Intel. StrongARM SA-1100 microprocessor developer's manual. `http://developer.intel.com/design/strong/manuals/278088.htm`.

[89] Intel. *Intel SpeedStep technology*, Jan 2000.

[90] Intel, Microsoft, and Toshiba. Advanced Configuration and Power Interface specification, revision 1.0b. `http://www.teleport.com/~acpi`, February 1999.

[91] IrDA. Infrared data association specifications. `http://http://www.irda.org/standards/specifications.asp`, 1994-2002.

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

[93] ISO. Open Distributed Processing- Reference Model, International Standard 10746-2 Itu-T Recommendation X.902. `http://citeseer.nj.nec.com/3915.html`, 1995.

[94] A.P. Chandrakasan J. Goodman. An energy-efficient reconfigurable public-key cryptography processor. *IEEE Journal of Solid-State Circuits*, pages 1808–1820, November 2001.

[95] A. Smith J. Lorch. Software strategies for portable computer energy management. *IEEE Personal Communications*, June 1998.

[96] M.B. Jones, D. Rosu, and M.-C. Rosu. Cpu reservations and time constraints: Efficient, predictable scheduling of independent activities. In *Symposium on Operating Systems Principles*, pages 198–211, 1997.

[97] A.R. Karlin, M.S. Manasse, L.A. McGeoch, and S.S. Owicki. Competitive randomized algorithms for nonuniform problems. *Algorithmica*, 11(6):542–571, June 1994.

[98] I. Kasai, Y. Tanijiri, T. Endo, and H. Ueda. A forgettable near eye display. In *IEEE International Conference on Wearable Computing (ISWC)*, October 2000.

[99] S.J. Kerry. Ieee 802.11 wireless lan specifications. `http://grouper.ieee.org/groups/802/11/`.

[100] K. Kim and K. Nahrstedt. A resource broker model with integrated reservation scheme. In *IEEE International Conference on Multimedia and Expo (II)*, pages 859–862, 2000.

[101] J. Kin, M. Gupta, and W.H. Mangione-Smith. The filter cache: An energy efficient memory structure. In *International Symposium on Microarchitecture*, pages 184–193, 1997.

[102] K.T. Kornegay, G. Qu, and M. Potkonjak. Quality of service and system design. In *IEEE Workshop on VLSI 99*, pages 112–117, 1999.

[103] C.E. Kozyrakis and D.A. Patterson. A new direction for computer architecture research. *IEEE Computer*, 31(11):24–32, 1998.

[104] R. Kraemer and M. Methfessel. A vertical approach to energy management. In *European Wireless 99 & ITG Mobile Communications conference*, Munchen, Germany, October 1999.

[105] R. Kravets and P. Krishnan. Power management techniques for mobile communication. In *4th Int. Conf. on Mobile Computing and Networking (MOBICOM)*, pages 157–168, 1998.

[106] P. Krishnan, P. Lon, and J Vitter. Adaptive disk spindown via optimal rent-to-buy in probabilistic environments. *Algorithmica*, (23):31–56, 1999.

[107] T. Kuroda, K. Suzuki, S. Mita, T. Fujita, F. Yamane, F. Sano, A. Chiba, Y. Watanabe, K. Matsuda, T. Maeda, T. Sakurai, and T. Furuyama. Variable supply-voltage scheme for low-power high-speed CMOS digital design. *IEEE Journal of Solid-State Circuits*, 33(3), March 1998.

[108] J. Kymissis, C. Kendall, J.A. Paradiso, and N. Gershenfeld. Parasitic power harvesting in shoes. In *Second IEEE International Conference on Wearable Computing (ISWC)*, pages 132–139, 1998.

[109] R.L. Lagendijk. Ubiquitous Communications updated technical annex. Technical Report 2000/1, Ubiquitous Communications, 2000.

[110] N. Langstraat. A model and runtime algorithms for forward-looking benefit-based resource allocation. Master's thesis, Delft University of Technology, 2001.

[111] D. Lee. Energy management issues for computer systems. `http://citeseer.nj.nec.com/434764.html`.

[112] Y. Lee and C. Krishna. Voltage-clock scaling for low energy consumption in real-time embedded systems. In *6th Int. Conf. on Real-Time Computing Systems and Applications*, 1998.

[113] B. Li. *Agilos: A Middleware Control Architecture for Application-Aware Quality of Service Adaptations*. PhD thesis, University of Illinois at Urbana-Champaign, May 2000.

[114] K. Li, R. Kumpf, P. Horton, and T. Anderson. Quantitative analysis of disk drive power management in portable computers. In *the 1994 Winter USENIX conference*, January 1994.

[115] Y.-T.S. Li, S. Malik, and A. Wolfe. Performance estimation of embedded software with instruction cache modeling. *Design Automation of Electronic Systems*, 4(3):257–279, 1999.

[116] Y.-R. Lin, C.-T. Hwang, and A.C.-H. Wu. Scheduling techniques for variable voltage low-power design. *ACM Trans. on Design Automation of Electronic Systems*, 2(2):81–97, Apr 1997.

[117] D. Linden. *Handbook of Batteries and Fuel Cells*. McGraw-Hill, New York, 1984.

[118] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard real time environment. *Journal of the ACM*, 20(1):46–61, January 1973.

[119] J. Lorch. The complete picture of the energy consumption of a portable computer. Master's thesis, UC Berkeley, December 1995.

[120] J.R. Lorch and A.J. Smith. Improving dynamic voltage scaling algorithms with pace. In *Sigmetrics 2001*, Cambridge, Massachusetts, USA, June 2001.

[121] Y.-H. Lu, L. Benini, and G. De Micheli. Operating-system directed power reduction. In *International Symposium on Low Power Electronics and Design*, pages 37–42. Stanford University, July 2000.

[122] Y.-H. Lu, L. Benini, and G. De Micheli. Requester-aware power reduction. In *International Symposium on System Synthesis (ISSS)*, September 2000.

[123] Y.-H. Lu, L. Benini, and G. De Michelli. Power-aware operating systems for interactive systems. *IEEE transactions on very large scale integration (VLSI) systems*, 10(2), April 2002.

[124] Y.-H. Lu and G. De Micheli. Adaptive hard disk power management on personal computers. *IEEE Great Lakes Symposium on VLSI*, pages 50–53, 1999.

[125] Y.-H. Lu and G. De Micheli. Comparing system-level power management policies. *IEEE Design and Test of Computers*, 18(2), March 2001.

[126] Y.-H. Lu, T. Simunic, and G. De Micheli. Software controlled power management. In *7th International workshop on Hardware/Software Codesign*, May 1999.

[127] M. Chiodo, P. Giusto, H. Hsieh, A. Jurecska, L. Lavango, and A. Sangiovanni-Vincentelli. A formal specification model for hardware/software codesign. In *In Proceeding of International Workshop on Hardware-Software Codesign*, 1993.

[128] M. Manasse, L. McGeoch, and D. Sleator. Competitive algorithms for on-line problems. In ACM, editor, *Proceedings of the twentieth annual ACM Symposium on Theory of Computing*, pages 322–333, New York, NY, USA, 1988. ACM Press.

[129] A. Manzak and C. Chakrabarti. Variable voltage task scheduling for minimizing energy or minimizing power. In *IEEE Int. Conf. on Acoustic, Speech, and Signal Processing (ICASSP'00)*, pages 3239 –3242, June 2000.

[130] T. Martin. *Balancing batteries, power, and performance: system issues in CPU speed-setting for mobile computing*. PhD thesis, Carnegie Mellon University, August 1999.

[131] M. Mattavelli and S. Brunetton. Implementing real-time video decoding on multimedia processors by complexity prediction techniques. *IEEE Transactions on Consumer Electronics*, 44(3):760–767, August 1998.

[132] V. McConnell. Green power aluminum/air fuel cell boosts cell phone use time to 8 hours. In *Scientific American monthly fuel cell industry report*, volume 1, December 2000.

[133] Sun Microsystem's. Sun microsystem's enterprise java beans technology site. `http://java.sun.com/products/ejb/`.

[134] MikkoM and Uriyan. Dll-hell. `http://www.wikipedia.com/wiki/DLL-hell`.

[135] J. Monteiro, S. Devadas, P. Ashar, and A. Mauskar. Scheduling techniques to enable power management. In *Design Automation Conference*, pages 349–352, 1996.

[136] K. Nahrstedt and J.M. Smith. The QOS broker. *IEEE Multimedia*, 2(1):53–67, 1995.

[137] T. Natsuno. *i-mode Strategy - Why Doesn't the World Catch Up?* 2000. ISBN 4931466281.

[138] B. Noble. System support for mobile, adaptive applications. *IEEE Personal Communications*, February 2000.

[139] B.D. Noble, M. Satyanarayanan, D. Narayanan, J.E. Tilton, J. Flinn, and K.R. Walker. Agile application-aware adaptation for mobility. In *16th ACM Symposium on Operating System Principles*, St. Malo, France, October 1997.

[140] Nokia. Nokia expects solid revenue growth next year as industry enters new phase. `http://press.nokia.com/PR/200111/841753_5.html`, November 2001.

[141] J.K. Ousterhout. Scheduling Techniques for Concurrent Systems. In *Third International Conference on Distributed Computing Systems*, pages 22–30, May 1982.

[142] D. Pan. Digital audio compression. *Digital Technical Journal of Digital Equipment Corporation*, 5(2):28–33, Spring 1993.

[143] D. Pan. A tutorial on MPEG/audio compression. *IEEE MultiMedia*, 2(2):60–74, 1995.

[144] D. Panigrahi, C. Chiasserini, S. Dey, R. Rao, A. Raghunathan, and K. Lahiri. Battery life estimation of mobile embedded systems. In *14th International Conference on VLSI Design (VLSID 2001)*, 2001.

[145] T. Pering, T. Burd, and R. Brodersen. The simulation and evaluation of dynamic voltage scaling algorithms. In *International Symposium on Low Power Electronics and Design*, August 1998.

[146] T. Pering, T. Burd, and R. Brodersen. Voltage scheduling in the lpARM microprocessor system. In *International Symposium on Low Power Electronics and Design*, July 2000.

[147] A. Petrick and J. Zyren. Tutorial on basic link budget analysis. `http://www.intersil.com/data/an/an9/an9804/an9804.pdf`, June 1998.

[148] P. Pillai and K.G. Shin. Real-time dynamic voltage scaling for low-power embedded operating systems. In *18th ACM Symposium on Operating Systems Principles*, 2001.

[149] J. Pouwelse, K. Langendoen, R. Lagendijk, and H. Sips. Power-aware video decoding. In *22nd Picture Coding Symposium*, Seoul, Korea, April 2001.

[150] J. Pouwelse, K. Langendoen, and H. Sips. Power consumption trade-offs for wireless audio access. In *Int. Workshop on Mobile Multimedia Communications (MoMuC 2000)*, pages P.6.1–P.6.6, Tokyo, Japan, October 2000.

[151] J. Pouwelse, K. Langendoen, and H. Sips. Dynamic voltage scaling on a low-power microprocessor. In *7th ACM Int. Conf. on Mobile Computing and Networking (Mobicom)*, pages 251–259, Rome, Italy, July 2001.

[152] J. Pouwelse, K. Langendoen, and H. Sips. Energy priority scheduling for variable voltage processors. In *Int. Symposium on Low Power Electronics and Design (ISLPED'01)*, Huntington Beach, CA, August 2001.

[153] J. Pouwelse, K. Langendoen, and H. Sips. Application-directed voltage scaling. *IEEE Transactions on VLSI Systems*, 11(5), October 2003.

[154] G. Qu and M. Potkonjak. Energy minimization with guaranteed quality of service. In *International Symposium on Low Power Electronics and Design*, pages 43–48, 2000.

[155] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek. A resource allocation model for QoS management. In *IEEE Real-Time Systems Symposium*, pages 298–307, December 1997.

[156] R. Rajkumar, C. Lee, J.P. Lehoczky, and D.P. Siewiorek. Practical solutions for QoS-based resource allocation. In *IEEE Real-Time Systems Symposium*, pages 296–306, 1998.

[157] D. Ramanathan, S. Irani, and R. Gupta. An analysis of system level power management algorithms and their effects on latency. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, March 2000.

[158] K. Rijkse. H.263: video coding for low-bit-rate communication. *IEEE Communications Magazine*, 34, December 1996.

[159] S. Russel and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 1995.

[160] B. Sabata, S. Chatterjee, M. Davis, J.J. Sydir, and T.F. Lawrence. Taxonomy for QoS specifications. In *Workshop on Object-Oriented Real-Time Dependable Systems (WORDS)*. IEEE Computer Society, 1997.

[161] J.H. Saltzer, D.P. Reed, and D.D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[162] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, pages 10–17, August 2001.

[163] O. Schelen, A. Nilsson, J. Norrgard, and S. Pink. Performance of QoS agents for provisioning network resources. In *the 7th international IWQoS conference*, London, UK, June 1999.

[164] B.N. Schilit, M.M. Theimer, and B.B. Welch. Customizing mobile application. In *USENIX Symposium on Mobile and Location-independent Computing*, pages 129–138, Cambridge, MA, US, 1993.

[165] A. Schmidt, M. Beigl, and H.-W. Gellersen. There is more to context than location. *Computers and Graphics*, 23(6):893–901, 1999.

[166] C.E. Shannon and W. Weaver. *The Mathematical Theory of Communication*. The University of Illinois Press, Urbana, 1964.

[167] B.J. Sheu, D.L. Scharfetter, P.-K. Ko, and M.-C. Jeng. Bsim: Berkeley short-channel igfet model for mos transistrors. *IEEE Journal of Solid-State Circuits*, SC-22(4):558–563, 1987.

[168] D. Shin, S. Lee, and J. Kim. Intra-task voltage scheduling for low-energy hard real-time applications. In *IEEE Design & Test of Computers*, March 2001.

[169] Y. Shin and K. Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Design Automation Conference*, pages 134–139, 1999.

[170] Y. Shin, K. Choi, and T. Sakurai. Power optimization of real-time embedded systems on variable speed processors. In *ICCAD*, November 2000.

[171] T. Simunic, L. Benini, A. Acquaviva, P. Glynn, and G. De Micheli. Dynamic voltage scaling for portable systems. In *Design Automation Conference*, 2001.

[172] A. Sinha and A.P. Chandrakasan. Energy aware software. In *13th International Conference on VLSI Design*, January 2000.

[173] H. So and A. Woo. A simple energy saving scheme on pda's using hardware scheduled dvs. Technical Report Class Report CS252, University of California, Berkeley, Department of Computer Science, December 1998.

[174] Bluetooth special interest group. Bluetooth specifications version 1.1. `http://www.bluetooth.com/dev/specifications.asp`.

[175] P. Stanley-Marbell, M. Hsiao, and U. Kremer. A hardware architecture for dynamic performance and energy adaptation. In *In, PACS-02, in Conjuntion 8th International Symposium on High-Performance Computer Architecture , HPCA-8.*, February 2002.

[176] T. Starner. Human-powered wearable computing. *IBM Systems Journal*, 35(3/4):618–629, 1996.

[177] P. Steenkiste. Adaptation models for network-aware distributed computations. In *3rd Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing (CANPC'99)*, pages 16–31, 1999.

[178] M. Stemm and R.H. Katz. Measuring and reducing energy consumption of network interfaces in hand-held devices. *IEICE Transactions on Communications*, E80-B(8):1125–31, 1997.

[179] N. Stratford and R. Mortier. An economic approach to adaptive resource management. In *7th Workshop on Hot Topics in Operating Systems (HotOS-VII)*, pages 142–147, Rio Rico, AZ, March 1999.

[180] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: a first step towards software power minimization. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 2(4):437–445, 1994.

[181] Transmeta-corporation. The technology behind the Crusoe processor. `http://www.transmeta.com/pdf/white_papers/paper_aklaiber_19jan00.pdf`.

[182] T.E. Truman, T. Pering, R. Doering, and R.W. Brodersen. The infopad multimedia terminal: a portable device for wireless information access. *IEEE Transactions on Computers*, 47(10):1073–1087, 1998.

[183] Ubiquitous Communications. The Ubicom home page. `http://www.ubicom.tudelft.nl`, January 1998.

[184] S. Udani and J. Smith. The power broker: Intelligent power management for mobile computers. Technical Report MS-CIS-96-12, Department of Computer Science, University of Pennsylvania, May 1996.

[185] T. Šimunić, L. Benini, and G.D. Micheli. Event-driven power management of portable systems. In *International Symposium on System Synthesis*, pages 18–23, 1999.

[186] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced CPU energy. In *Symposium on Operating Systems Design and Implementation (OSDI)*, pages 13–23, November 1994.

[187] J. Wilkes. Predictive power conservation. Technical Report Technical Report HPL-CSP-92-5, Hewlett-Packard Laboratories, February 1992.

[188] G. Wilson and M.A. Sasse. Do users always know whats good for them? utilising physiological responses to assess media quality. In *HCI 2000*, pages 327–339, Sunderland, UK, September 2000.

[189] Winbond. Wts701 text-to-speech chip data sheet. `http://www.winbond-usa.com/products/`, 2001. Winbond Electronics Corporation America.

[190] H. Woesner, J.-P. Ebert, M. Schlager, and A. Wolisz. Power saving mechanisms in emerging standards for wireless LANs: The MAC level perspecitve. *IEEE Personal Communications*, 5(3):40–48, June 1998.

[191] L.C. Wolf, L. Delgrossi, R. Steinmetz, S. Schaller, and H. Wittig. Issues of reserving resources in advance. *Lecture Notes in Computer Science*, 1018, 1995.

[192] X. Xiao and L.M. Ni. Internet QoS: A big picture. *IEEE Network*, 13(2):8–18, March 1999.

[193] D. Xu, K. Nahrstedt, A. Viswanathan, and D. Wichadakul. QoS and contention-aware multi-resource reservation. *Cluster Computing, the Journal of Networks, Software Tools and Applications*, 2001.

[194] F. Yao, A. Demers, and S. Shenker. A scheduling model for reduced CPU energy. In *36th IEEE Symposium on Foundations of Computer Science*, pages 374–382, October 1995.

[195] C. Young, N.C. Gloy, and M.D. Smith. A comparative analysis of schemes for correlated branch prediction. In *ISCA*, pages 276–286, 1995.

[196] W. Yuan and K. Nahrstedt. A middleware framework coordinating processor/power resource management for multimedia applications. In *Proceedings of IEEE Globecom 2001*, 2001.

[197] W. Yuan, K. Nahrstedt, and X. Gu. Coordinating energy-aware adaptation of multimedia applications and hardware resource. In *9th ACM Multimedia (Multimedia Middleware Workshop)*, pages 60–63, October 2001.

[198] H. Zeng, C. Ellis, A. Lebeck, and A. Vahdat. Currentcy: Unifying policies for resource management. Technical Report TR CS-2002-09, May 2002.

[199] H. Zeng, X. Fan, C. Ellis, A. Lebeck, and A. Vahdat. ECOSystem: Managing energy as a first class operating system resource. In *Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, October 2002.

[200] L. Zhang, S. Deering, and D. Estrin. RSVP: A new resource ReSerVation protocol. *IEEE network*, 7(5):8–18, September 1993.

Acknowledgments

This Ph.D. thesis is the result of research conducted within the Ubicom project at TUDelft. This thesis is obviously not the result of my own individual efforts alone, but the fruit of collaboration with many people, both at TUDelft and TNO-FEL.

I want to thank my family and my friends for supporting me in my Ph.D. and for continuously showing interest in my work.

Without the contribution of Jan-derk Bakker and Eric Mouw this thesis would be lacking a significant amount of depth in the experiments. Also thanks to Ivaylo for the help during power measurements and use of his testing environment.

Johan Pouwelse,
1 March 2003,
Delft.

Samenvatting

Het stroomverbruik van kleine computers is een groot probleem aan het worden. Binnen handzame computers en mobiele telefoons moeten meer gegevens worden uitgewisseld tussen hardware, het besturingssysteem en de bovenliggende software. In dit werk presenteren we een architectuur die dit mogelijk maakt. De efficiëntie van deze architectuur is zo hoog dat er tientallen procenten minder stroom worden verbruikt en er nieuwe functionele mogelijkheden ontstaan zoals instelbare batterij levensduur door middel van adaptiviteit.

Curriculum Vitae

Johan Pouwelse was born in Middelburg, the Netherlands, on December 19th 1972. In 1992 he finished the engineering school at mid-level. The same year he started studying Computer Science at The Hague Polytechnic school. He carried out his B.Sc. project at Technolution N.V. in Gouda, the Netherlands were he created a system for low speed data communication across the 230 V power mains. Software for this system was made, including an error correcting communication protocol. After three and a halve years of study he recieved his B.Sc. with a specialisation in data communication.

He started a M.Sc. study at Delft University of Technology in January 1996 with a specialisation in distributed systems. His master project topic was on performance measurements of TCP/IP across satellite links. A working system was build of two computers with TCP/IP, ATM switches, satcom modems and an analog satellite simulator. After only two years of study he recieved a Cum Laude degree at Delft University of Technology.

From February 1998 until Jan 2003 he worked at the UbiCom program of Delft University of Technology, sponsored by the Dutch organisation for applied science (TNO). For this project he worked closely with more then two dozen researchers in the field of augmented reality, compiler design, electronics, radio propagation, and information theory. During the UbiCom program he worked for four months in a laboratory in Miyazakiday, Japan on UbiCom topics.

Currently he works at the Cactus project of Delft University of Technology. This project investigates wireless ad-hoc networks and peer-to-peer networks. During the summer of 2003 he was a visiting scientist at the peer-to-peer group of the Massachusetts Institute of Technology (MIT), Boston, USA.