DYNAMIC POWER MANAGEMENT OF ELECTRONIC SYSTEMS

GIOVANNI DE MICHELI
Stanford University
Stanford, CA 94305

LUCA BENINI
DEIS - Università di Bologna
Bologna, Italy

AND

ALESSANDRO BOGLIOLO
DEIS - Università di Bologna
Bologna, Italy

Abstract— Dynamic power management is a design methodology aiming at controlling performance and power levels of digital circuits and systems, with the goal of extending the autonomous operation time of battery-powered systems, providing graceful performance degradation when supply energy is limited, and adapting power dissipation to satisfy environmental constraints.
We survey dynamic power management applied at the system level. We analyze first idleness detection and shutdown mechanisms for idle hardware resources. We review industrial standards for operating system-based power management, such as the Advanced Configuration and Power Interface (ACPI) standard proposed by Intel, Microsoft and Toshiba. Next, we review system-level modeling techniques, and describe stochastic models for the power/performance behavior of systems. We analyze different modeling assumptions and we discuss their validity. Last, we describe a method for determining optimum policies and validation methods, via simulation at different abstraction levels, for power managed systems.

1. Introduction

Design methodologies for energy-efficient system-level design are receiving an increasingly larger attention. The motivations for such interest are rooted in the widespread use of portable electronic appliances (e.g., cellular phones, laptop computers, etc.) and in the concerns about the environmental impact of electronic systems (whether mobile or not). In the former case, low-power circuit and system design are required to provide a reasonable operation time to battery-operated devices. In the latter case, heat dissipation may pose a practical limitation to the design and use of high-performance processors. Moreover, studies [19] have shown that present computers consume a significant amount of electric energy, and a corresponding amount of non-renewable energy resources. Thus system-level design must strike the balance between providing high service levels to the users while curtailing power dissipation. In other words, we need to increase the energetic efficiency of electronic systems, as it has been done, by other means, with other types of engines.

Electronic systems are *heterogeneous* in nature, by combining digital with analog circuitry, using semiconductor (e.g., RAM, FLASH memories) and electro-mechanical (e.g., disks) storage resources, as well as electro-optical (e.g., displays) human interfaces. Power management must address all types of resources in a system. The power breakdown for a well-known laptop computer [32] shows that, on average, 36% of the total power is consumed by the display, 18% by the *hard-disk drive* (HDD), 18% by the wireless *local area network* (LAN) interface, 7% by non-critical components (keyboard, mouse etc.), and only 21% by digital VLSI circuitry, mainly memory and *central processing unit* (CPU). Reducing the power in the digital components of this laptop by *10X* would reduce the overall power consumption by less than 19%.

Lowering system-level power consumption, while preserving adequate service and performance levels, is a difficult task. Indeed, reducing system performance (e.g., by using lower clock rates) is not a desirable option when considering the increasingly more elaborate software application programs for computers and features of portable electronic devices. On the other hand, present systems have several components which are not utilized at all times. When such components are *idle*, they can be put in sleep states with reduced (or null) power consumption, with a limited (or null) impact on performance.

Dynamic power management is a design methodology aiming at controlling performance and power levels of digital circuits and systems, by exploiting the idleness of their components. A system is provided with a *power manager* that monitors the overall system and component states and

controls the state transitions. The control procedure is called power management *policy*. We believe that dynamic power management is the most appropriate approach to reduce power consumption under performance constraints, because significant power waste is associated with idle resources and because of its general applicability. Note that support for dynamic power management must be provided by the overall system organization, and system architects often envision system partitions that enable power management. Despite the fact that some systems are designed with power management schemes, *computer-aided design* (CAD) support for this task is limited, if any at all exists. Thus, designing and implementing dynamic power management schemes are usually manual tasks.

The power management policy plays a key role in determining the efficacy of a power managed system. This chapter describes some computational methods for the determination of policies that are optimum under some system modeling assumptions. Management policies can be implemented in hardware or in software. In the case of simple systems, it may be best to implement policies as specialized hardware control units that can be modeled as *finite-state machines* (FSMs). Such units monitor the system components' states to determine their evolution in time.

When considering programmable digital systems, or generic systems with a programmable digital component, it is possible to migrate to software the task of controlling the power states. In particular, the *operating system* (OS) is the software layer where the policy can be implemented best, for those systems, like computers, that have an OS. *OS-based power management* (OSPM) has the advantage that the power/performance dynamic control is performed by the software layer (the OS) that manages the computational, storage and I/O tasks of the system. Implementing OSPM is a *hardware/software co-design* problem, because the hardware resources need to be interfaced with the OS-based software power manager, and because both the hardware resources and the software application programs need to be designed so that they cooperate with OSPM.

Recent initiatives to handle system-level power management include Microsoft's *OnNow* initiative [36] and the *Advanced Configuration and Power Interface* (ACPI) standard proposed by Intel, Microsoft and Toshiba [34]. The former supports the implementation of OSPM and targets the design of personal computers with improved usability through innovative OS design. The latter simplifies the co-design of OSPM by providing an interface standard to control system resources. On the other hand, the aforementioned standards do not provide procedures for optimal control of power-managed system.

System-level dynamic power management can be complemented by specific chip-level design techniques for power reduction. Low-power consump-

tion in integrated circuits can be achieved through the combination of different techniques, including architectural design choices [7], logic and physical design [18, 23], choice of circuit families and implementation technology [26]. In particular, dynamic power management can be applied to digital circuits by specific techniques, such as supply voltage, frequency and activity control. Examples of activity control are clock gating of control units [2] and signal gating of data-path [16] units. Such approaches are based on the principle of exploiting idleness of circuits or portions thereof. They involve both detection of idle conditions and the freezing of power-consuming activities in the idle components. We refer the interested reader to [1] for a comprehensive and comparative description of dynamic power management methods at the chip and system level.

In this chapter, we survey system-level dynamic power management. We consider first system-level design issues, such as idleness detection and shutdown mechanisms for idle resources. We review the OnNow and ACPI standards, as well as previous work in the area of power management.

Next, we review system-level modeling techniques, and introduce stochastic models for the power/performance behavior of systems. We analyze different modeling assumptions and we discuss their validity. We consider then a working model, for which optimal policies can be computed. We present next how policies can be implemented in electronic systems.

Last but not least, we describe several methods for validating the policies, based on simulation at different abstraction levels. We conclude by stressing the need for CAD tools to support model identification, policy optimization and validation for dynamically power-managed systems.

2. System design

In this section we consider issues related to system-level design. We view the system hardware as a collection of resources, we characterize their idleness and present methods for their shutdown. We consider then the interface standards that support resource monitoring and control from the operating system, and we review current related work on dynamic power management.

2.1. IDLENESS AND SHUTDOWN MECHANISMS

The basic principle of a dynamic power manager is to detect inactivity of a resource and shut it down. A fundamental premise is that the idleness detection and power management circuit consumes a negligible fraction of the total power.

We classify idleness as *external* or *internal*. The former is strongly tight to the concept of observability of a resource's outputs, while the latter can be related to the notion of internal state, when the resource has one. A

circuit is externally idle if its outputs are not required during a period of time. During such period, the resource is functionally redundant and can be shut down, thus reducing power consumption. A resource is internally idle when it produces the same output over a period of time. Thus, the outputs can be stored and the resource shut down.

While external idleness is a general concept applicable to all types of resources (e.g., digital, analog, memories, hard-disks, displays), internal idleness is typical of digital circuits. Thus, we will be concerned with external idleness detection and exploitation, since we address here system-level design.

There are several mechanisms for shutting down a resource. Digital circuits can be "frozen" by disabling registers (by lowering the enable input) or by gating the clock. By freezing the information on registers, data propagation through combinational logic is halted, with a corresponding power saving. (This saving may be significant in CMOS static technologies, where power is consumed mainly during transitions).

A radical approach to shutdown is to turn off power to a resource. While this mechanism is conceptually simple and applicable in general, it usually involves a non-negligible time to restore operation. Note that in some cases the context must be saved before shutdown (e.g., in non-volatile memory) and restored at restart.

Some components can be shut down at different levels, each one corresponding to a power consumption level and to a delay to restore operation. As a first example, consider a backlit display. When the display is used, both the LCD array and the backlighting are on. When the user is idle, the backlighting and/or the LCD array can be turned off with different power savings.

As a second example, a hard-disk drive [37] may have an operational state, in addition to an idle, a low-power idle, a standby, and a sleep state. In the idle states the disk is spinning, but some of the electronic components of the drive are turned off. The transition from idle to active is extremely fast, but only 50-70% of the power is saved in these states. In the standby and sleep states, the disk is spun down, thus reducing power consumption by 90-95%. On the other hand, the transition to the active state is not only slow, but it causes additional power consumption, because of the acceleration of the disk motor.

This example shows the trade-off of power versus performance in dynamic power management. The lower the power associated with a system state, the longer the delay in restoring an operational state. Dynamic power management strategies need to take advantage of the low-power states while minimizing the impact on performance.

2.2. INDUSTRIAL DESIGN STANDARDS

Industrial standards have been proposed to facilitate the development of operating system-based power management. A precursor standard is *Advanced Power Management* (APM) [35], which provides several layers of software to support power management on computers with compliant resources. APM defines the software interface between an OS power management policy driver and software for hardware-specific power management. APM partitions the power management functionality into a hierarchy of cooperating layers and standardizes the flow of information and control among them. With this scheme, APM-compliant application software issues commands to an APM driver which control the resources through the BIOS layer [1].

More recently, Intel, Microsoft and Toshiba proposed the *Advanced Configuration and Power Interface* (ACPI) [34], described in detail in the next section, which superseeds APM. In contrast to APM, which hinges upon the BIOS code layer, ACPI provides an OS-independent power management and configuration standard. It provides for an orderly transition from *legacy* hardware to ACPI-compliant hardware. Although this initiative targets *personal computers* (PCs), it contains useful guidelines for a more general class of systems. The characterizing feature of ACPI is that it recognizes dynamic power management as the key to reducing overall system power consumption, and it focuses on making the implementation of dynamic power management schemes in personal computers as straightforward as possible.

The ACPI specification forms the foundation of the *OnNow initiative* [36] launched by the Microsoft Corporation. The OnNow initiative is specific to the design of personal computers (PCs) and proposes the migration of power management algorithms and policies into the computer's operating system (OS). The scope of OnNow goes beyond dynamic power management.

An OnNow-compliant PC platform must conform to a set of requirements [36], including:

- The PC is ready for use as soon as the user turns it on.
- The PC appears as off when not in use, but it must be capable of responding to wake-up events (originated by the user or by a resource, such as a modem sensing an incoming call).
- Software tracks hardware status changes and adjusts accordingly. OS and software applications cooperate in the frame of dynamic power

[1] The *basic input output system* (BIOS) is the lowest layer of the OS that is often customized to the hardware.

management: applications are aware that resources may be in different service states and release resources when they are unneeded.
– All hardware devices participate in the power management scheme, by responding to the OS commands.

Personal computers are just beginning to meet the requirements of OnNow in 1998. The migration of power management to the operating system level will yield a profound improvement of the performance, power consumption and quality of service of personal computers, because it will give the control of the system to the component (i.e., the OS) that can make the most informed decisions. OnNow relies on the ACPI infrastructure to interface the software to the hardware components to be managed.

2.2.1. *ACPI*

ACPI [34] is an OS-independent, general specification that applies to desktop, mobile and home computers as well as to high-performance servers. The specification has emerged as an evolution of previous initiatives that attempted to integrate power management features in the low-level routines that directly interact with hardware devices (firmware and BIOS). It also provides some form of *backward compatibility* since it allows ACPI-compliant hardware resources to co-exist with legacy non-ACPI-compliant hardware.

ACPI is the key element for implementing operating system power management strategies, such as *OnNow*. It is an open standard that is made available for adoption by hardware vendors and operating system developers. The main goals of ACPI are to:

– Enable all PCs to implement motherboard dynamic configuration and power management.
– Enhance power management features and the robustness of power managed systems.
– Accelerate implementation of power-managed computers, reduce costs and time to market.

The ACPI specification defines most interfaces between OS software and hardware. The software and hardware components relevant to ACPI are shown in Figure 1. Applications interact with the OS kernel through *application programming interfaces* (APIs). A module of the OS implements the power management policies. The power management module interacts with the hardware through kernel services (system calls). The kernel interacts with the hardware using device drivers. The front-end of the ACPI interface is the *ACPI driver*. The driver is OS-specific, it maps kernel requests to ACPI commands, and ACPI responses/messages to kernel signals/interrupts. Notice that the kernel may also interact with non-ACPI-compliant hardware through other device drivers.

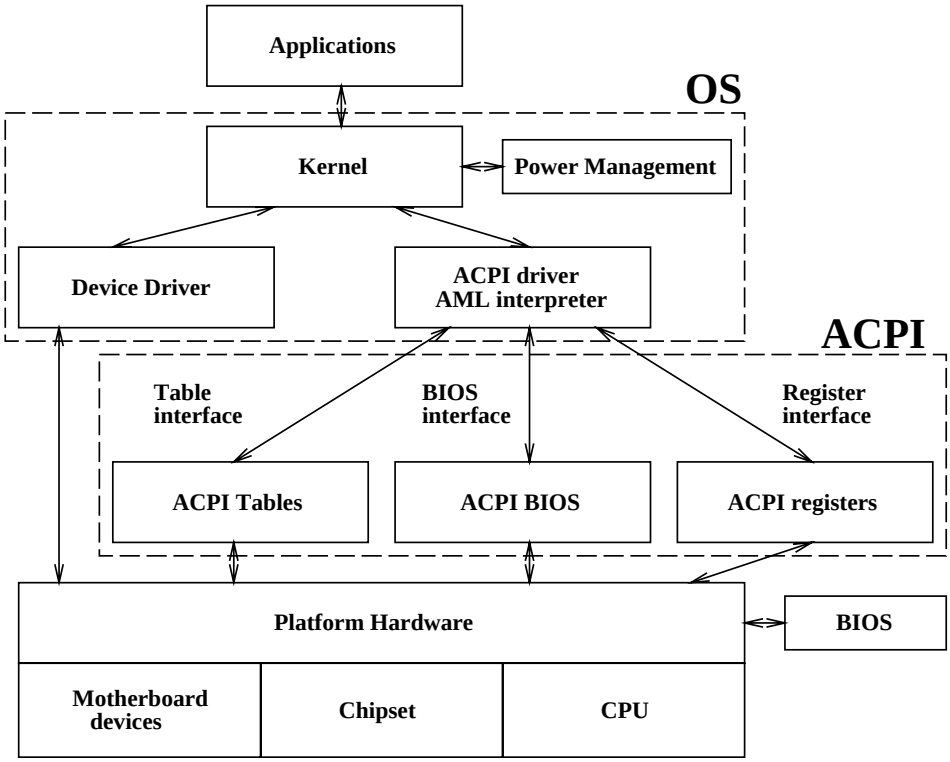

Figure 1. ACPI interface and PC platform

At the bottom of Figure 1 the hardware platform is shown. Although it is represented as a monolithic block, it is useful to distinguish three types of hardware components. First, hardware resources (or *devices*) are the system components that provide some kind of specialized functionality (e.g., video controllers, modems, bus controllers). Second, the *CPU* can be seen as a specialized resource that need to be active for the OS (and the ACPI interface layer) to run. Finally, the *chipset* (also called core logic) is the motherboard logic that controls the most basic hardware functionalities (such as real-time clocks, interrupt signals, processor busses) and interfaces the CPU with all other devices. Although the CPU runs the OS, no system activity could be performed without the chipset. From the power management standpoint, the chipset, or a critical part of it, should always be active, because the system relies on it to exit from sleep states.

It is important to notice that ACPI specifies neither how to implement hardware devices nor how to realize power management in the operating system. No constraints are imposed on implementation styles for hardware and on power management policies. Implementation of ACPI-compliant

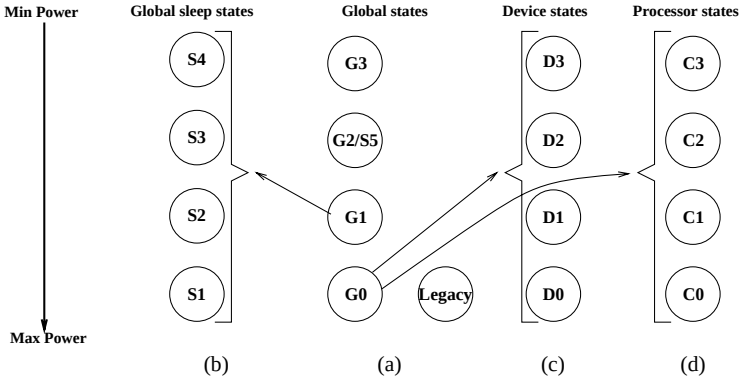

Figure 2. State definitions for ACPI

hardware can leverage any technology or architectural optimization as long as the power-managed device is controllable by the standard interface specified by ACPI.

In ACPI, the system has five *global power states*. Namely:

- *Mechanical off* state $G3$, with no power consumption.
- *Soft off* state $G2$ (also called $S5$). A full OS reboot is needed to restore the working state.
- *Sleeping* state $G1$. The system appears to be off and power consumption is reduced. The system returns to the working state in an amount of time which grows with the inverse of the power consumption.
- *Working* state $G0$, where the system is ON and fully usable.
- *Legacy* state, which is entered when the system does not comply with ACPI.

The global states are shown in Figure 2 (a). They are ordered from top to bottom by increasing power dissipation.

The ACPI specification refines the classification of global system states by defining four sleeping states within state $G1$, as shown in Figure 2 (b):

- $S1$ is a sleeping state with low wake-up latency. No system context is lost in the CPU or the chipset.
- $S2$ is a low wake-up latency sleeping state. This state is similar to the $S1$ sleeping state with the exception that the CPU and system cache context is lost.
- $S3$ is another low wake-up latency sleeping state where all system context is lost except system memory.
- $S4$ is the sleeping state with the lowest power and longest wake-up latency. To reduce power to a minimum, all devices are powered off.

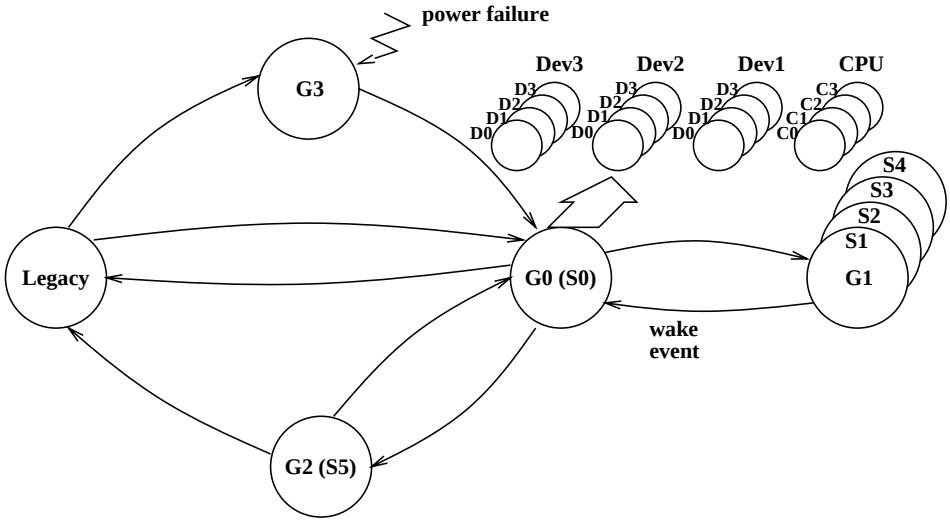

Figure 3. Global and power states and substates

Additionally, the ACPI specification defines states for system components. There are two types of system components, *devices* and *processor*, for which power states are specified. Devices are abstract representations of the hardware resources in the system. Four states are defined for devices, as shown in Figure 2 (c). In contrast with global power states, device power states are not visible to the user. For instance, some devices can be in an inactive state, but the system appears to be in a working state. Furthermore, state transitions for different devices can be controlled by different power management schemes.

The processor is the central processing unit that controls the entire PC platform. The processor has its own power states, as shown in Figure 2 (d). Notice the intrinsic asymmetry of the ACPI model. The central role of the CPU is recognized, and the processor is not treated as a simple resource.

Special devices are *embedded controllers*, that function as resources for the main CPU. ACPI defines a specialized interface for embedded controllers. Although from a power management point of view embedded controllers are treated as normal resources, they have specialized drivers because they may be used to monitor power-related system characteristics, perform low-level complex calculations, and they may provide data that is required to implement power management policies. For example, an embedded controller can be used to control board temperature sensors and provide valuable data for thermal management.

States and transitions for an ACPI-compliant system are shown in Figure 3. Usually the system alternates between the working ($G0$) and the

sleeping ($G1$) states. When the entire system is idle or the user has pressed the power-off button, the OS will drive the computer into one of the states on the left side of Figure 3. From the user's viewpoint, no computation occurs. The sleeping sub-states differ in which *wake* events can force a transition into a working state, and how long the transition should take. If the only wake-up event of interest is the activation of the user turn-on button and a latency of a few minutes can be tolerated, the OS could save the entire system context into non-volatile storage and transition the hardware into a soft-off state ($G2$). In this state, power dissipation is almost null and context is retained (in non-volatile memory) for an arbitrary period of time. The mechanical off state ($G3$) is entered in the case of power failure or mechanical disconnection of power supply. Complete OS boot is required to exit the mechanical off state. Finally, the *legacy* state is entered in case the hardware does not support OSPM.

It is important to note that ACPI provides only a framework for designers to implement power management strategies, while the the choice of power management *policy* is left to the engineer.

2.3. RELATED WORK

We consider now work in different areas related to dynamic power management. The common theme is the search of methods for power/performance management. Techniques and application domains vary widely.

Chip-level power management features have been implemented in mainstream commercial microprocessors [9, 10, 11, 14, 29]. Microprocessor power management has two main flavors. First, the entire chip can be put in one of several sleep states through external signals or software control. Second, chip units can be shut down by stopping their local clock distribution. This is done automatically by dedicated on-chip control logic, without user control. Techniques for the automatic synthesis of chip-level power management logic are thoroughly surveyed in [1].

At a much higher level of abstraction, energy-conscious communication protocols based on power management have been extensively studied [17, 25, 27, 33]. The main purpose of these protocols is to regulate the access of several communication devices to a shared medium trying to obtain maximum power efficiency for a given throughput requirement.

Power efficiency is a stringent constraint for mobile communication devices. Pagers are probably the first example of mobile device for personal communication. In [17], communication protocols for pagers are surveyed. These protocols have been designed for maximum power efficiency. Protocol power efficiency is achieved essentially by increasing the fraction of time in which a single pager is idle and can operate in a low-power sleep state

without the risk of loosing messages.

Golding et al. considered HDD sub-systems [12, 13], and presented an extensive study of the performance of various disk spin-down policies. The problem of deciding when to spin down a hard disk to reduce its power dissipation is presented as a variation of the general problem of predicting idleness for a system or a system component. This problem has been extensively studied in the past by computer architects and operating system designers (reference [13] contains numerous pointers to work in this field), because idleness prediction can be exploited to optimize performance (for instance by exploiting long idle period to perform work that will probably be useful in the future). When low power dissipation is the target, idleness prediction is employed to decide when it is convenient to spin down a disk to save power (if a long idle period is predicted), and to decide when to turn it on (if the predictor estimates that the end of the idle period is approaching).

The studies presented in [28, 15] target hypothetical "interactive terminals". A common conclusions in these works is that future workloads can be predicted by examining the past history. The prediction results can then be used to decide when and how transitioning the system to a sleep state. In [28], the distribution of idle and busy periods for the interactive terminal is represented as a time series, and approximated with a least-squares regression model. The regression model is used for predicting the duration of future idle periods. A simplified power management policy is also introduced, that predicts the duration of an idle period based on the duration of the last activity period. The authors of [28] claim that the simple policy performs almost as well as the complex regression model, and it is much easier to implement. In [15], an improvement over the simple prediction algorithm of [28] is presented, where idleness prediction is based on a weighted sum of the duration of past idle periods, with geometrically decaying weights. The weighted sum policy is augmented by a technique that reduces the likelihood of multiple mispredictions.

A common feature of these power management approaches is that policies are formulated heuristically, then tested with simulations or measurements to assess their effectiveness.

3. Modeling

In the sequel we consider the hardware part of the system as a set of resources. We model the resources at a very-high level of abstraction, i.e., we view them as units that perform or request specific services and that communicate by requesting and acknowledging such services. Resources of interest are, of course, those that can be power managed, i.e., those that

can be set in different states, as in the ACPI scheme.

From a power management standpoint, we model the hardware behavior as a *finite-state system*, where each resource is associated with a set of states and can be in one of the corresponding states. Power and service levels are associated with the different states and transitions among states. In this modeling style, we abstract away the functionality of the resource, and we are concerned only with the ability of the resource to provide and/or request a service.

Because of the high-level of abstraction in resource modeling, it is difficult, if not impossible, to have precise information about power and performance levels of each resource. This uncertainty can be modeled by using random variables for the observable quantities of interest (eg., power, performance), and by considering average values as well as their statistical distributions [1]. This stochastic approach is required to capture both the non-determinism due to lack of detailed information in the abstract resource models as well as the fluctuations of the observed variables due to environmental factors.

With this modeling style, computing optimum dynamic power management policies becomes a *stochastic optimum control* problem [6]. The problem solution, and its accuracy in modeling reality, depend highly on the assumptions we use in modeling. We will discuss next the impact of some modeling assumptions, and then consider is detail a system model under some specific assumption that enables us to compute optimum policies, as shown in Section 4.

3.1. MODELING ASSUMPTIONS

A system model can be characterized by the ensemble of its components, their mode of interaction and their statistical properties.

In general, we can view resources both as providers and requesters of services to other resources. In practice, some resources will be limited to providing or requesting services. We call *system structure* the system abstraction where resources are vertices of a directed graph and where resource interaction is shown by edges. The interaction is the request of a service and/or its delivery. Special resources, such as *queues*, can be used to model the accumulation or requests waiting for services [31].

A simple example of a CPU requesting data to a hard-disk drive is shown in Figure 4 (a). A more complex example is reported in Figure 4 (b): it shows a CPU interacting with a LAN interface, a HDD, a display, a keyboard and a mouse. Requests to the CPU can be originated from the keyboard, mouse and LAN interface. Requests to the display come from the CPU (which also forwards requests from the keyboard and mouse). The

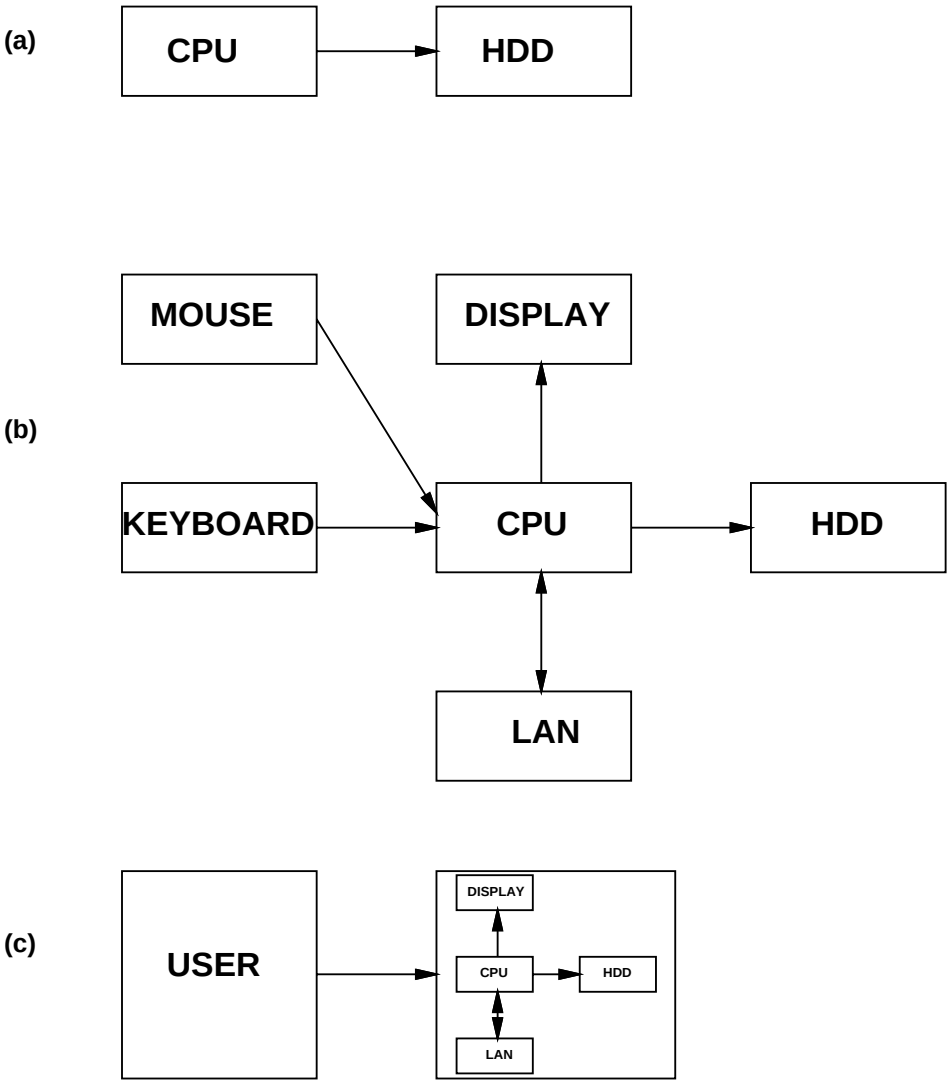

Figure 4. (a) CPU requesting data to a HDD. (b) Simple model of some resources of a personal computer and their interaction. (c) User-PC model where the requests sent by the keboard and by the mouse are lumped as a single requester and the CPU, HDD, LAN and display are lumped as a single provider.

CPU can request services to the HDD and LAN. Note that the keyboard and mouse models can express also the behavior of the human user who hits their keys and buttons.

A modeling trade-off exists in the *granularity* of the resources, i.e., between the number and average complexity of the resources. Whereas a

system model with several resources and an associated structure can capture the interaction of the system components in a detailed way, most researchers view systems with a very coarse granularity. Namely, systems are identified by one resource providing a service, called *service provider*, and one unit requesting a service, called *service requester*. The requester models the *workload source*. This granularity can be used to model systems like user-PC, as shown in Figure 4 (c), where the keyboard and mouse are lumped as a single requester and the CPU, HDD and LAN are seen as a single provider.

Let us consider now the statistical properties of the components of a system. *Stationarity* of a stochastic process means that its statistical properties are invariant to a shift of the time origin [22]. When resources are viewed as providers of services in response to input stimuli, it is conceivable to model their behavior as stationary. Conversely, when resources act as workload sources, and when we model users' requests as such, the stationarity assumption may not hold in general. For example, patterns of human behavior may change with time, especially when considering the fact that an electronic system may have different users. On the other hand, observations of workload sources over a wide time interval may lead to stationary models that are adequately accurate. An advantage of using stationary models is the relative ease of solving the corresponding stochastic optimization problems.

The statistical properties of each component are captured by their distributions. An important aspect is the statistical independence (or dependence) of the resources' statistical models from each others. When a system structure can be captured by disjoint graphs corresponding to statistically-independent resources, the system decomposition allows us to consider and solve independent subproblems. In practice, weak dependencies can sometimes be neglected. Conversely, system structures with many dependencies correspond to complex models requiring a large computational effort to solve the related optimization problems. As a result, the identification of the system resources, interactions and statistics is a crucial step in modeling real systems.

3.2. A WORKING MODEL

We consider here a working model, with one provider that receives requests through a queue, and that is controlled by a power manager (PM), as shown in Figure 5. This model is described in more detailed in [21]. We summarize here the salient features of the model.

We assume stationary stochastic models for a service provider (SP), a service requester (SR), and a queue (Q). We assume also that the service

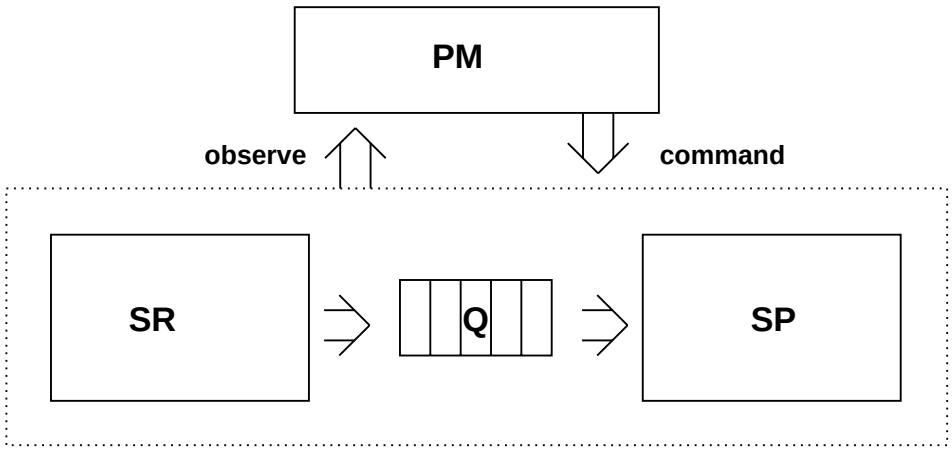

Figure 5. Coarse-grained system model

requester is statistically independent from the other components. We consider a discrete-time setting, i.e., we divide time into equally-spaced time slices. We use a parametrized Markov chain model to represent the statistical properties of the system resources. By using the Markov assumption, transition probabilities depend only on the current state and not on the previous history. Moreover, we assume that transition probabilities depend on a parameter, that models the command issued by the power manager. We consider next the system components in detail.

Service provider. It is a device (e.g., HDD) which services incoming requests from a workload source. In each time interval, it can be in only one *state*. Each state $s_p \in \{1, 2, \ldots, S_p\}$ is characterized by a performance level and by a power-consumption level. In the simplest case, we could have two states ($S_p = 2$): *on* and *off*. Otherwise, the states may be more, and in particular match states (and substates) as defined by the ACPI standard. At each timepoint, transitions between states are controlled by a power manager through *commands* $a \in A = \{1, 2, \ldots, N_a\}$. For example, we can define two simple commands: switch on (*s_on*) and switch off (*s_off*). When a specific command is issued, the SP will move to a new state at the next timepoint with a fixed probability dependent only on the command a itself, and on the departure and arrival states. In other terms, after being given a transition command by the power manager, the SP can remain in its current state during the next time slice with a non-zero probability. This aspect of the model takes into account the uncertainty in the transition time between states caused by the abstraction of functional information. Our probabilistic model is equivalent to the assumption that the evolu-

tion in time of states is modeled by a Markov process that depends on the commands issued by the power manager. Each state has a specific *power consumption rate*, which is function both of the state and the command issued. The SP provides service in one state only, that we call active state.

Service requester. It sends requests to the SP. The SR is modeled as a Markov chain, whose state corresponds to the number of requests s_r (with $s_r \in \{0, 1, \ldots, S_r - 1\}$) sent to the SR during time slice of interest.

Queue. It buffers incoming service requests. We define its length to be $(S_q - 1)$. The queue length is also Markov process with state $s_q \in \{0, 1, \ldots, S_q\}$. The state of the queue depends on the state of the provider and requester, as well as on the command issued by the power manager in the time slice of interest.

Power manager. It communicates with the service provider and attempts to set its state at each timepoint, by issuing commands chosen among a finite set A. For example, the commands can be *s_on*, and *s_off*. The power manager contains all proper specifications and collects all relevant information (by observing SP and SR) needed for implementing a power management policy. The consumption of the power manager is assumed to be much smaller than the consumption of the subsystems it controls and it is not a concern here.

The state of the system consisting of {SP,SR,Q} and managed by PM is a triple $s = (s_r, s_p, s_q)$. Being the composition of three Markov chains, s is a Markov chain (with $S = S_r \times S_p \times S_q$ states), whose transition matrix depends on the command a issued by the PM.

Let us consider a simple example, as shown in Figure 6, representing a power-managed HDD. The service requester has only two states, 0 and 1, representing the number of requests per time slice sent to the provider. The queue of the service provider has two states, 0 and 1, representing the number of requests to be serviced. The service provider has two states, *on* and *off*, representing its functional state. When *on*, it services up to one request per time slice taken from the queue. The corresponding power consumption is of 3W. When *off* it does not service any request and it consumes no power. However, a power consumption of 4W is associated with any transition between the two states. SR evolves independently, while the transition probabilities of SP depend on the command issued by the power manager (*s_on, s_off*) and those of the queue depend on the states of both SP and SR, as well as on the command. For example, consider the SP in state *on*. (Center-left of Figure 6.) When command *s_on* is issued,

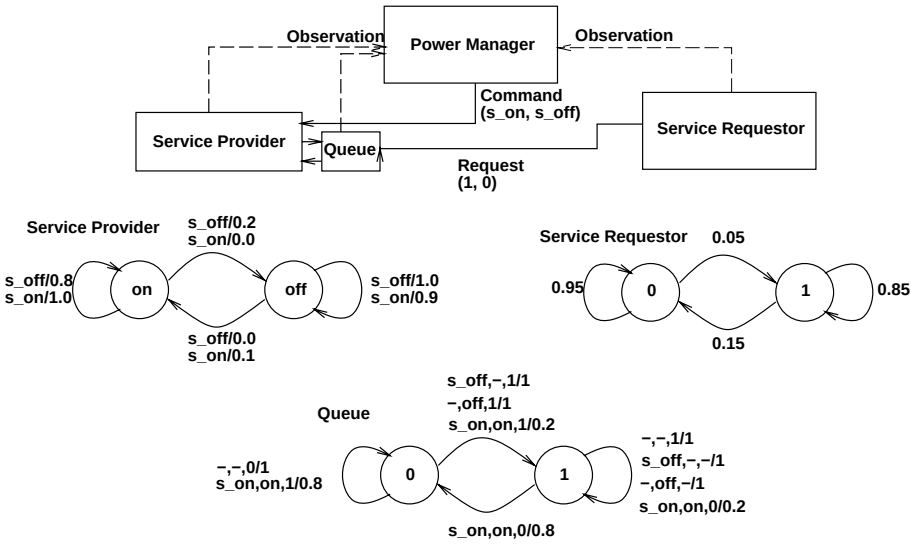

Figure 6. An example of a system model with one service provider, one service requester and one queue,with corresponding Markov chains.

the SP will stay in state *on* with probability 1, and transit to state *off* with probability 0. Conversely, when command *s_off* is issued, it will stay in state *on* with probability 0.8, and transit to state *off* with probability 0.2.

3.3. EXTENSIONS AND LIMITATIONS

System providers, requesters and queues with several internal states can be modeled in a straightforward way. Power costs and performance penalties can be associated with states and transitions of the Markov models. Thus, the simple model exemplified by Figure 6 can be made more detailed, to capture subtle differences among resource states (e.g., discriminating *soft off* states from *sleeping* states).

Similarly, more complex system structures (with multiple providers, requesters and queues) can be modeled by considering the combined effect of the resources' models. This can be easily done under the hypothesis of statistical independence of the resources' behavior, as in the case of several independent providers responding to a single workload source. In this particular case the overall system model can be derived by composing the Markov chains associated with each resource.

Unfortunately, in the general case, the system model is not amenable to a simple decomposition. Consider for example systems such as the one depicted in Figure 4 (b). The interaction among components causes statistical dependence. Most requests to the display from the CPU are triggered

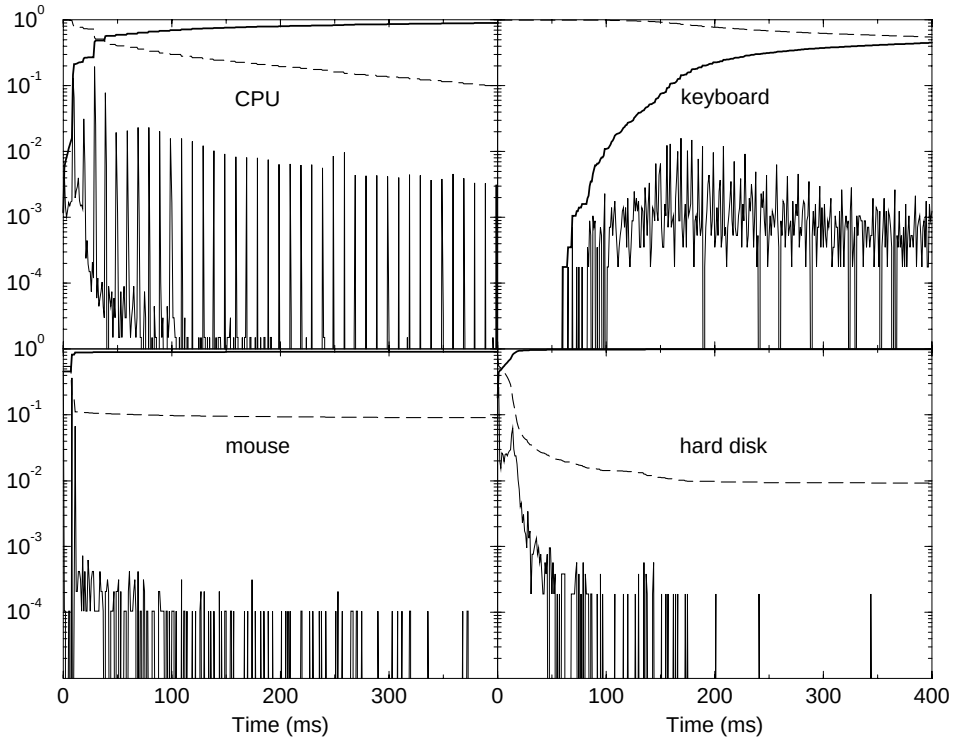

Figure 7. Statistical analysis of the inter-arrival times between service requests for CPU, keyboard, mouse and HDD of a personal computer during software development. For each device, three curves are plotted in lin-log scale: the probability density (solid line), the probability distribution (bold line) and its complement to 1 (dashed line).

by the mouse and keyboard. Thus it is not possible to view the resources as having an independent behavior.

Even when considering systems with simple structures, the identification of the statistical distributions is not a simple matter. The use of stationary Markov models corresponds to use geometric distributions for requests and service times. Such a model may deviate from reality. For example, resources may have known, deterministic service delays compounded with non-deterministic delays depending on the environment.

3.4. EXTRACTING MODELS FOR THE USER

System users can be viewed as workload sources and modeled as service requesters. An approach to model the user behavior consists of *monitoring* the system during a user session and then *extracting* a statistical model of his/her behavior.

System monitoring has to be sufficiently accurate to provide time-stamped traces of service requests. The cumulative counts provided by the system utilities of many computer systems are not sufficient to steer power management. In addition, monitoring has to be non-perturbative in order to affect usage patterns as little as possible. A monitoring system specifically designed for supporting dynamic power management in personal computers is described in reference [4]: the prototype implementation is conceived as an extension of the Linux operating system [30]. The monitoring tool can be configured to collect information about many resources at the same time. Measured overhead for data collection is quite small (around 0.4%). Figure 7 shows usage statistics simultaneously extracted for the CPU, the keyboard, the mouse and the HDD of a personal computer during one-hour of software development.

Once time-stamped request traces have been collected, they are used to characterize the abstract model for the SR. If a discrete-time setting is assumed for modeling, the trace need to be discretized first. For a given time step T, that is usually of the same order of the minimum time constant of the SP, a discretized trace is a stream of integer numbers representing request counts. The k-th number in the stream (i.e., n_k) is the number of requests with time stamps in the interval $[(k-1)\cdot T, k\cdot T]$. According to the definition of SR proposed in Section 3.2, n_k represents the state of the SR at the k-th time step. Characterizing a Markov model for the user consists of tuning the state transition probabilities in order to make the statistical properties of the model as similar as possible to those of the stream. To this purpose, state transition probabilities are directly computed from the discretized trace. For instance, the probability associated with the transition from state $s_r = 0$ to state $s_r = 1$ is obtained as the ratio between the number of $0,1$ sequences in the stream and the total number of 0's. This procedure extracts the most accurate Markov model of any trace. If the trace cannot be seen as the output of a Markov process, then the SR Markov model needs to be validated by simulation, as described in Section 5.

4. Policy optimization

We consider now the policy optimization problem, for the working model described in Section 3.2. Policy optimization strives at minimizing the average power consumption under performance constraints. Similarly, we can define the complementary optimization of maximizing system performance under a bound on the average power consumption. With the working model of Section 3.2, performance relates to the average delay in servicing a request (i.e., wait time on a hard-disk access). Due to space limitation, we

describe only the major steps toward solving the problem. The interested reader is referred to [21] for details.

We need to analyze first how the PM controls the system, to define formally the notion of policy, which is the unknown of the problem to optimize.

At each time point, the power manager observes the history of the system and controls the SP by taking a *decision*. A *deterministic decision* consists in issuing a single command. A *randomized decision* consists of specifying the probability of issuing a command. Randomized decisions include deterministic decisions as special cases (i.e., the probability of a command is 1).

A *policy* is a finite sequence of decisions. A *stationary policy* is one where the same decision (as a function of the system state) is taken at each time point. Note that stationarity means that the functional dependency of the decision on the state does not change over time. Obviously, as the state evolves, the decisions change. *Markov stationary policies* are policies where decisions depend only on the present system state.

The importance of Markov stationary policies stems from two facts: they are easy to implement and it is possible to show that optimum policies belong to this class. Namely, it is possible to prove formally that the aforementioned policy optimization problems have an optimum solution that is a unique randomized Markov stationary policy. In the particular case that either the problem is unconstrained or the constraints are inactive, then the solution is also deterministic [6, 21]. It is possible to show that the policy optimization problem can be cast as a linear program. An intuitive formulation is described here in an informal way. Consider the PM, that observes the system state and issues commands. For each possible pair (state,command), we can compute its *frequency*, i.e., the expected number of times that a system is in that state and issues that command. The frequency is a non-negative number subject to the following *conservation law*. The expected number of times state x is the current state is equal to the expected initial population of x plus the expected number of times x is reached from any other state. Moreover, average power and performance loss can be expressed as linear functions of the $(state, command)$ frequencies. Thus, minimizing power consumption can be expressed as minimizing a linear function of the $(state, command)$ frequencies, under linear constraints.

Overall, linear programs modeling policy optimization can be efficiently solved by standard software packages, for simple topologies and a reasonable number of commands. The policy optimization tool described in [21] is built around *PCx*, an advanced LP solver based on an interior point algorithm [8].

Figure 8 shows the power-performance trade-off curve obtained for the

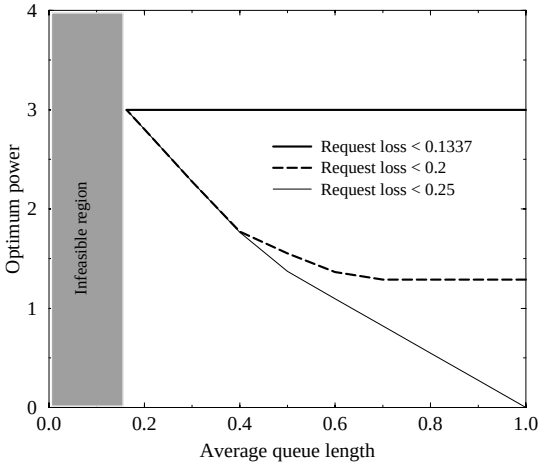

Figure 8. Power-performance trade-off curves for the example system of Figure 6.

example system of Figure 6 by iteratively solving the policy optimization problem for different performance constraints. Performance is expressed in term of average queue length, that is the average waiting time for a request. An additional constraint is used, called *request loss*, to represent the maximum probability of loosing a request because of a queue-full condition. It is worth noting how the power-performance trade-off is affected by the additional constraint. In particular, if a request-loss lower than 0.1337 has to be guaranteed, the SP can never be shut down. In this case, no power savings can be achieved regardless of the performance constraint.

The trade-off curve for a more complex system is reported in Figure 9. The SP is a commercially-available power-manageable HDD with one active state and four inactive states, spanning the trade off between power consumption and shut-down/wake-up times [39]. The average power consumption of the disk when in the active state is of 2.5W. The SR model was extracted as described in Section 3.4 from the time-stamped traces of disk accesses provided in [38]. A queue of length 2 was used.

Points associated with several heuristic policies are also plotted in the power-performance plane for comparison. Although we cannot claim that our heuristic policies are the best that any experienced designer can formulate, some of them provide power-performance points not far from the trade-off curve. Note that heuristic solutions do not allow the designer to automatically take constraints into account. On the other hand, trial and error approaches may be highly expensive due to the large number of pa-

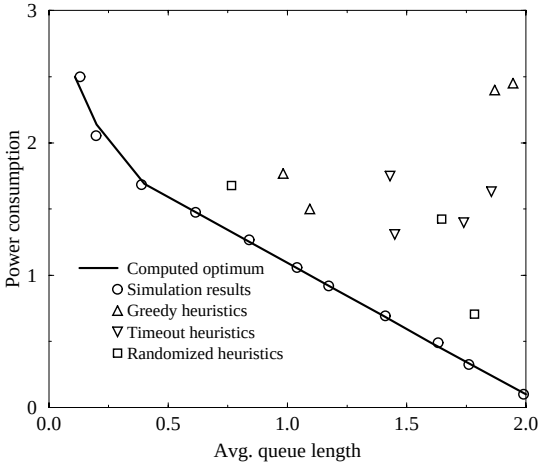

Figure 9. Power-performance feasible trade-off's for a commercially-available power-manageable hard disk.

rameters (in our case study the policy is represented by a 66x5 matrix with 330 entries). Moreover, even if it is possible to produce heuristic policies that produce "reasonable" results, there is no way for the designer to estimate if the results can be improved. For these reasons, computer-aided design tools for policy optimization can be of great help to system designers.

4.1. POWER MANAGER IMPLEMENTATION

Power management policies can be computed *off-line* or *on-line*. In the former case, a policy is computed once for all for the system being designed, and implemented in hardware or software as described in this section. Alternatively, several policies can be computed off-line and stored, each corresponding to a different environmental factor, such as a workload source. The power manager can switch among the policies at run time. On-line policy computation is also possible. Once the power manager has identified a change of the environmental conditions that make the current policy no longer effective, a new policy can be computed which takes into account the new environmental parameters (e.g., request arrival rate). Once the policy is computed, it can be executed until the power manager deems it appropriate.

In the case of simple systems, it may be practical to implement the dynamic power management policy as a hardware control circuit. Since circuit synthesis methods are currently used for hardware design, policy

implementation consists of representing the policy in a synthesizable *hardware description language* (HDL) model for the power manager. In general, the circuit input is the system state and the output are the commands.

Deterministic policies can be implemented by table look-up schemes. Randomized policies require storing the conditional probabilities of issuing a command in any given state and comparing them with a pseudo-random number, which can be generated by using a linear feedback shift register (LFSR). The command probabilities should be normalized to the length of the LFSR. In particular, when only two commands are possible (e.g., *s_on* and *s_off*), their conditional probabilities sum up to 1 and thus only one probability needs to be stored. The binary outcome of the comparison with a pseudo-random number corresponds to the chosen command. This scheme can be easily extended to handle N_a commands by means of a table with $N_a - 1$ entries per state and $N_a - 1$ comparisons with the pseudo-random number, which can be executed in parallel.

The implementation of policies in software requires the software synthesis of the power manager (e.g., the generation of a C program that issues the commands as a function of the system state) as well as its embedding in the operating system. In the case of randomized policies, the program should make use of a pseudo-random number generator for deciding which command should be issued. The power manager may be executed in kernel mode and be synchronized and/or merged with the OS task scheduler to reduce the performance penalty due to context switch.

5. Validation

In this section we address the problem of bridging the gap between the high level of abstraction at which policy optimization is performed and the real-world systems, where optimal policies have to be applied. In Section 3 we have described a general approach for modeling power-manageable systems as interacting Markov processes. In Section 4 we have shown that such an abstract model allows us to cast the policy optimization problem as a linear program that can be solved in polynomial time. All modeling assumptions made to formulate and solve the policy optimization task need to be tested in order to validate its results. We briefly describe validation techniques based on simulation and emulation at different abstraction levels, ranging from the direct simulation of the Markov models used for optimization to the actual implementation of the optimal policies in the target systems. We discuss the main strengths and the inherent limitations of each approach.

Discrete-time simulation of Markov processes. Discrete-time simulation is performed at the same abstraction level used for optimization. The simula-

tor takes the policy and the Markov models of the components and iteratively performs the following steps: *i*) take a decision (based on the current state), *ii*) evaluate cost/performance metrics, *iii*) evaluate the next state of all components, *iv*) increment time, update the state and iterate. Notice that both the policy and the next-state functions of the Markov chains are *non-deterministic discrete functions* (NDFs): inputs are present-state variables and commands, whereas outputs are the outcomes of random processes. NDFs can be represented as matrices having as many rows as input configurations and as many columns as output values. Entries represent the conditional probabilities of all possible outcomes for all given input configurations. To evaluate a function, the row associated with the current input configuration is selected and a pseudo-random number (uniformly distributed between 0 and 1) is generated and compared to the entries in the row to select the actual command.

Needless to say, this simulation paradigm cannot be used to validate the policy against the modeling assumptions of Section 3, since it relies on them as well. However, it provides valuable information about the time-domain system behavior. Constraints and objective functions used for optimization are average expected values of the performance/cost metrics of interest. Simulation allows us to monitor the instantaneous values of such parameters (to detect, for instance, the temporary violations of performance constraints) and to measure their variance.

Discrete-time simulation with actual user traces. The simulation paradigm is the same described in the previous paragraph. The only difference is that the model of the service requester is now replaced by a trace taken from a real-world application. At each time step, the present state of the SR is read from the trace, instead of being non-deterministically computed from the previous one.

Though the abstraction level is still very high, trace simulation allows us to remove all assumptions on the time distribution of service requests. As a result, it can be used to check the validity of the Markov model used for the SR during optimization.

Discrete-time simulation with real request traces was performed to validate the trade-off curve of Figure 9. Simulation results are denoted by circles in figure. The small distance of the circles from the solid-line curve is a measure of the quality of the SR Markov model extracted from the user traces and used for optimization.

Event-driven stochastic simulation. In event-driven simulation, model evaluation is no longer periodic. The model of each component is re-evaluated only when an event (i.e., a change) occurs on some of the state/command

variables it depends on. The evaluation of a component may produce new events to be scheduled at a future time. Both the output events and their scheduling times may be non-deterministic. For instance, the command issued by a randomized policy can be modeled as an instantaneous non-deterministic event, while the transition between two states of the SP can be viewed as a deterministic event (if the next state is uniquely determined once a command has been issued) to be scheduled at a non-deterministic time (if the transition time is a random variable). The scheduling time is pseudo-randomly chosen according to a given probability distribution. An event-driven stochastic simulator is described in [5].

The main advantage of the event-driven paradigm is that it can easily handle stochastic processes with arbitrary distributions. Conversely, discrete-time simulation is implicitly based on the memory-less assumption that is behind Markov models, that allows us to represent and simulate only geometrically-distributed random variables. Adding memory information to a Markov model in order to represent different distributions is not a practical solution since it causes the exponential increase of the number of states. Event-driven simulation provides a more practical way of applying optimal policies to arbitrary SP models in order to check the validity of the Markov model used for optimization.

Fully-functional simulation. The functionality of a system can be described at many levels of abstraction. Functional simulation can be performed at any level. Here we focus on *cycle-accurate* simulation, that is the most accurate simulation paradigm that can be used to handle systems as complex as a personal computer. Cycle-accurate simulation matches the behavior of the real system at clock boundaries. When the system is a computer, cycle-accurate simulation provides enough detail to boot an operating system and run an actual workload on top of it. A fully-functional simulator specifically designed to study computer systems is *SimOS* [24], that can handle multi-processor architectures and provides models for simulating commercial microprocessors, peripherals and operating systems.

When system functionality comes into the picture, most of the simplifying modeling assumptions can be eliminated. In particular, stochastic models for SP and SR are no longer required, since even their functionality can be exactly simulated. Performance penalties can be realistically estimated and accurate cost metrics (i.e., power consumptions) can be associated with the operating states of the resources. In addition, functional simulation realizes a unique trade-off between realism and flexibility. On one hand, it provides a means of validating the policies against the real world and gives the designer a direct hands-on experience of most of the implementation issues involved in OS-directed power management. On the

other hand, it allows the designer to explore the entire design space, balancing hardware and software solutions.

The main drawback of functional simulation is performance: simulation times may be more than three orders of magnitude slower than the run times on the corresponding real system, making the approach impractical to study complex workloads.

Emulation. We use the term *emulation* to denote a validation approach that uses functionally-equivalent real hardware components to exercise the behavior of part of the system. In particular, we are interested in using a computer without power-management features as the hardware platform to emulate a power-managed functionally-equivalent one. As an example, suppose that we are designing a power-management policy for the HDD of a laptop computer, having one active state and several inactive states (with different power consumptions and wake-up times). If such a HDD is not available for validation, the power-managed system can be emulated on an equivalent computer (with the same workload of the target one) with a non-power-manageable HDD. As long as the device used for emulation has the same performance of the target one, it can be employed to emulate the active-state functionality, while inactive states (and transitions between them) can be simulated by the software device driver. The code of the original device driver needs a few changes: *i*) an additional state variable representing the power state, *ii*) a routine for updating the power state according to power-management commands, *iii*) a timer to simulate state transition times, *iv*) a routine to provide power consumption estimates and *v*) a request-blocking mechanism that enables actual accesses to the disk only when in the active state. In general, emulation of power-managed systems is based on the observation that dynamic power management can only reduce system performances. Hence, if a functionally-equivalent real system is available for exercising the active-state performance, lower-performance states can be emulated as well.

Emulation has two desirable features. First, it runs at the same speed of the actual system, thus enabling policy validation against realistic workloads of any complexity and real-time interactive user sessions. This gives the user a direct experience of performance degradations possibly induced by power management. Second, it enables the software specification of the low-power states of the SP. The possibility of easily changing the SP model can be exploited both during the design of a power-manageable resource, to verify the effectiveness of a given low-power state, and during system-level design, to select among equivalent power-manageable components. The main drawback with respect to simulative approaches is that the sys-

tem architecture is assigned once for all: no architectural choices can be explored.

Implementation. Policies can be validated by testing their implementations. Since the policy is directly applied to the target system, its actual impact on the cost metrics of interest can be measured accurately. Thus experimentation at this level is useful as a final step in validating a given policy.

6. Conclusions

Dynamic power management is an effective means for system-level design of low-power electronic systems. Dynamic power management is already widely applied to system design, but today most electronic products rely on ad-hoc implementation frameworks (e.g., firmware code) and on heuristic management policies (e.g., timeout policies). We expect that the use of industrial standards, such as OnNow and ACPI, will soon facilitate the clean implementation of operating system based power management.

This survey has shown how systems can be modeled so that optimal management policies can be computed, validated and implemented. The computation of optimal policies is a new problem for system-level design. In particular, we have shown a working model for which the optimal stochastic power-management control problem can be efficiently and exactly solved. The solution method we have analyzed relies on a modeling abstraction of system resources in terms of Markov processes. Several extension can be made to the model, at the price of complicating the solution procedure, by considering more detailed system models. As in many design problems, good engineering judgment is key in determining the right balance among model accuracy, exactness of the solution (for the given model), and computational effort.

Due to the proliferation of handheld electronic systems, and due to increasingly stringent environmental constraints on non-mobile systems, we believe that designers will be very often confronted with the challenge of deriving optimal, or near optimal, dynamic power management solutions. As a result, computer-aided design tools for power management will be extremely useful is system-level design for model identification, policy optimization and validation.

7. Acknowledgments

We acknowledge support from NSF under contract MIP-942119. We would like to thank Eui-Young Chung, Yung Hsiang Lu, Giuseppe Paleologo and Tajana Simunic for several discussions on this topic.

References

1. L. Benini and G. De Micheli, *Dynamic Power Management of Circuits and Systems: Design Techniques and CAD Tools,* Kluwer, 1997.
2. L. Benini and G. De Micheli, "Automatic Synthesis of Low-Power Gated-Clock Finite-State Machines," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, no. 6, pp. 630–643, June. 1996.
3. L. Benini, P. Siegel and G. De Micheli, "Automatic Synthesis of Gated Clocks for Power Reduction in Sequential Circuits," *IEEE Design and Test of Computers*, pp. 32–40, Dec. 1994.
4. L. Benini, A.Bogliolo, S.Cavallucci and B. Riccò, "Monitoring System Activity for OS-directed Dynamic Power Management," *Proceedings International Symposium on Low-Power Design*, 1998.
5. L. Benini, R. Hodgson and P. Siegel, "System-Level Power Estimation and Optimization," to appear in *Proceedings of Int.l Symposium of Low-Power Electronics and Design*, 1998.
6. D. Bertsekas, "Stochastic Optimal Control," Academic Press, 1978.
7. A. Chandrakasan and R. Brodersen, *Low-Power Digital CMOS Design.* Kluwer, 1995.
8. J. Czyzyk, S. Mehrotra, and S. Wright, "PCx User Guide", *Technical Report OTC 96/01*, Optimization Technology Center, May, 1996.
9. G. Debnath, K. Debnath and R. Fernando, "The Pentium Processor-90/100, Microarchitecture and Low-Power Circuit Design," in *International conference on VLSI design*, pp. 185–190, Jan. 1995.
10. S. Furber, *ARM System Architecture* Addison-Wesley, 1997.
11. S. Gary, P. Ippolito et al., "PowerPC 603, a Microprocessor for Portable Computers," *IEEE Design & Test of Computers* vol. 11, no. 4, pp. 14-23, Win. 1994.
12. R. Golding, P. Bosch and J. Wilkes "Idleness is not Sloth", in *Proceedings of Winter USENIX Technical Conference*, pp. 201–212, 1995.
13. R Golding, P. Bosh and J. Wilkes, "Idleness is not Sloth" *HP Laboratories Technical Report* HPL-96-140, 1996.
14. M. Gowan, L. Biro, D. Jackson, "Power Considerations in the Design of the Alpha 21264 Microprocessor," *DAC - Proceedings of the Design Automation Conference*, 1998, pp. 726-731.
15. C.-H. Hwang and A. Wu, "A Predictive System Shutdown Method for Energy Saving of Event-Driven Computation", in *Proceedings of the Int.l Conference on Computer Aided Design*, pp. 28-32, 1997.
16. H. Kapadia, G. De Micheli and L. Benini, "Reducing Switching Activity on Datapath Buses with Control-Signal Gating," *CICC - Proceedings of the Custom Integrated Circuit Conference*, pp. 589-592, 1998.
17. B. Mangione-Smith, "Low-Power Communication Protocols: Paging and Beyond," *IEEE Symposium on Low-Power Electronics*, pp. 8–11, 1995.
18. J. Monteiro and S. Devadas, *Computer-Aided Techniques for Low-Power Sequential Logic Circuits.* Kluwer 1997.
19. B.Nadel, "The Green Machine," *PC Magazine*, Vol 12, No. 10, p.110, May 25, 1993.
20. W. Nebel and J. Mermet (Eds.), *Low-Power Design in Deep Submicron Electronics.* Kluwer, 1997.

21. G. Paleologo, L. Benini, A. Bogliolo and G. De Micheli, "Policy Optimization for Dynamic Power Management," *DAC - Proceedings of the Design Automation Conference*, 1998, pp. 182-187.
22. A. Papoulis, *Probability, Random Variables, and Stochastic Processes*, McGraw-Hill, 1984.
23. J. M. Rabaey and M. Pedram (editors), *Low-Power Design Methodologies.* Kluwer, 1996.
24. M. Rosenblum, E. Bugnion, S. Devine and S. A. Herrod, "Using the SimOS Machine Simulator to Study Complex Computer Systems," *ACM Transactions on Modeling and Computer Simulation*, Vol. 7, no. 1, pp. 78-103, 1997.
25. J. Rulnick and N. Bambos, "Mobile Power Management for Wireless Communication Networks," *Wireless Networks*, vol. 3, no. 1, pp. 3–14, 1997.
26. T. Sakurai and T. Kuroda, "Low-Power Circuit Design for Multimedia CMOS VLSI," in *Workshop on Synthesis and System Integration of Mixed Technologies*, pp. 3–10, Nov. 1996.
27. K. Sivalingham, M. Srivastava et al., "Low-power Access Protocols Based on Scheduling for Wireless and Mobile ATM Networks," *Int.l Conference on Universal Personal Communications*, pp. 429–433, 1997.
28. M. Srivastava, A. Chandrakasan. R. Brodersen, "Predictive System Shutdown and Other Architectural Techniques for Energy Efficient Programmable Computation," *IEEE Transactions on VLSI Systems*, vol. 4, no. 1, pp. 42–55, March 1996.
29. V. Tiwari, D. Singh, S. Rajgopal, G. Metha, R. Patel and F. Baez, "Reducing Power in High-Performance Microprocessors," *DAC - Proceedings of the Design Automation Conference*, 1998, pp. 732-737.
30. L. Torvalds, "Linux Kernel Implementation," *Proceedings of Open Systems. Looking into the future. AUUG'94*, pp. 9-14, 1994.
31. K. Trivedi, *Probability and Statistics with Reliability, Queueing and Computer Science Applications*, Prentice Hall, 1982.
32. S. Udani and J. Smith, "The Power Broker: Intelligent Power Management for Mobile Computing," *Technical report MS-CIS-96-12*, Dept. of Computer Information Science, University of Pennsylvania, May 1996.
33. M. Zorzi and R. Rao, "Energy-Constrained Error Control for Wireless Channels," *IEEE Personal Communications*, vol. 4, no. 6, pp. 27–33, Dec. 1997.
34. *http://www.intel.com/ial/powermgm/specs.html*, Intel, Microsoft and Toshiba, "Advanced Configuration and Power Interface specification", Dec. 1996.
35. *http://developer.intel.com/IAL/powermgm/apmovr.htm*, Intel, "Advanced Power Management Overview," 1998.
36. *http://www.microsoft.com/hwdev/pcfuture/ ONNOW.HTM*, Microsoft, "OnNow: the evolution of the PC platform," Aug. 1997.
37. *http://www.storage.ibm.com/storage/oem/data/ travvp.htm* Technical specification of hard-drive IBM Travelstar VP 2.5-inch, 1996.
38. *http://now.cs.berkeley.edu/Xfs/AuspexTraces/auspex.html*, Auspex File System Traces, 1993.
39. *http://www.storage.ibm.com/storage/oem/data/travvp.htm*, Hard Drive IBM Travelstar VP 2.5-inch, 1996.