A Survey of Design Techniques for System-Level Dynamic Power Management

Luca Benini, *Member, IEEE*, Alessandro Bogliolo, *Member, IEEE*, and Giovanni De Micheli, *Fellow, IEEE*

***Abstract*—*Dynamic power management* (DPM) is a design methodology for dynamically reconfiguring systems to provide the requested services and performance levels with a minimum number of active components or a minimum load on such components. DPM encompasses a set of techniques that achieves energy-efficient computation by selectively turning off (or reducing the performance of) system components when they are idle (or partially unexploited).**

In this paper, we survey several approaches to system-level dynamic power management. We first describe how systems employ power-manageable components and how the use of dynamic reconfiguration can impact the overall power consumption. We then analyze DPM implementation issues in electronic systems, and we survey recent initiatives in standardizing the hardware/software interface to enable software-controlled power management of hardware components.

***Index Terms*—Energy conservation, energy management, optimization methods.**

I. Introduction

MOST ELECTRONIC circuits and system designs are confronted with the problem of delivering high performance with a limited consumption of electric power. High performance is required by the increasingly complex applications (e.g., multimedia) that are running even on portable devices. Low-power consumption is required to achieve acceptable autonomy in battery-powered systems, as well as to reduce the environmental impact (e.g., heat dissipation, cooling-induced noise) and operation cost of stationary systems. In other words, achieving highly energy-efficient computation is a major challenge in electronic design.

Electronic systems can be viewed as collections of components, which may be heterogeneous in nature. Some components may have mechanical parts, e.g., hard-disk drives (HDD's), or optical parts, e.g., displays. For example, a cellular telephone has a digital very large scale integration (VLSI) component, an analog radio-frequency (RF) component, and a display. Such components may be active at different times, and correspondingly consume different fractions of the telephone power budget. Similarly, main components of portable computers are VLSI chips, HDD, and display. It is often the case that the HDD and the display are the most power-hungry components [1], and thus their effective use is key to achieving long operating times between battery recharges.

To be competitive, an electronic design must be able to deliver peak performance when requested. Nevertheless, peak performance is required only during some time intervals. Similarly, system components are not always required to be in the active state. The ability to enable and disable components, as well as of tuning their performance to the *workload* (e.g., user's requests), is key in achieving energy-efficient designs.

Dynamic power management (DPM) is a design methodology that dynamically reconfigures an electronic system to provide the requested services and performance levels with a minimum number of active components or a minimum load on such components [1], [2]. DPM encompasses a set of techniques that achieve energy-efficient computation by selectively turning off (or reducing the performance of) system components when they are *idle* (or partially unexploited). DPM is used in various forms in most portable (and some stationary) electronic designs; yet its application is sometimes primitive because its full potentials are still unexplored and because the complexity of interfacing heterogeneous components has limited designers to simple solutions.

The fundamental premise for the applicability of DPM is that systems (and their components) experience nonuniform workloads during operation time. Such an assumption is valid for most systems, both when considered in isolation and when internetworked. A second assumption of DPM is that it is possible to predict, with a certain degree of confidence, the fluctuations of workload. Workload observation and prediction should not consume significant energy.

Dynamic power managers can have different embodiments, according to the level (e.g., component, system, network) where DPM is applied and to the physical realization style (e.g., timer, hard-wired controller, software routine). Typically, a *power manager* (PM) implements a control procedure based on some observations and/or assumptions on the workload. The control procedure is often called *policy*. An example of a simple policy, ubiquitously used for laptops and palmtops, is the *timeout* policy, which shuts down a component after a fixed inactivity time, under the assumption that it is highly likely that a component remains idle if it has been idle for the timeout time. We shall show in this paper how this simple-minded policy may turn out to be inefficient and how it can be improved.

This paper has the objective to cover and relate different approaches to system-level DPM. We begin by describing how systems employ power-manageable components and how the

Manuscript received February 14, 1999; revised September 23, 1999. This work was supported in part by NSF under Contract CCR-9901190 and by the MARCO/DARPA Gigascale Silicon Research Center.

L. Benini is with the Dip. di Elettronica, Informatica e Sistemistica, Università di Bologna, Bologna 40136, Italy.

A. Bogliolo is with the Department of Engineering, Università di Ferrara, Ferrara 44100, Italy.

G. De Micheli is with the Computer Systems Laboratory, Stanford University, Stanford, CA 94305 USA.

Publisher Item Identifier S 1063-8210(00)04347-X.

use of their dynamic reconfiguration can impact the overall power consumption. Next, we review and compare different approaches to DPM. We use a mathematical framework to highlight the benefits and pitfalls of different power management policies. We classify power management approaches into two major classes, where policies are based on *predictive schemes* and *stochastic optimum control* respectively. Within each class, we survey the approaches being applied to system design and/or described in the literature. Last, we present the means of implementing DPM in electronic systems, and we describe in particular the recent initiatives in standardizing hardware/software interface to enable software-controlled power management of hardware components.

II. Modeling Power-Managed Systems

We model a power-managed system as a set of interacting *power manageable components* (PMC's) controlled by a *power manager* (PM). We model PMC's as *black boxes*. We are not concerned on how PMC's are designed (this topic will be deferred to Section IV), but we focus instead on how they interact with the environment. The purpose of this analysis is to understand what type and how much information should be exchanged between a power manager and system components in order to implement effective policies. We take a bottom-up view. We consider PMC's in isolation first. Then we describe DPM for systems with several interacting components. Finally, we analyze the problem of managing power for a *network* of communicating systems.

A. Power Manageable Components

Our working definition of *component* is general and abstract. A component is an atomic block in a complete system. Notice that the granularity of this definition is arbitrary, hence components can be as simple as a functional unit within a chip, or as complex as a board. The characterizing property of our definition is *atomicity*. At the system level, a component is seen as an indivisible functional block: no detailed knowledge of its internal structure is assumed. The fundamental characteristic of a PMC is the availability of multiple *modes of operation* that span the power-performance tradeoff. Nonmanageable components are designed for a given performance target and power budget. In contrast, with PMC's it is possible to dynamically switch between high-performance high-power modes of operation and low-power low-performance ones.

It is possible to think that a PMC may have a continuous range of operation modes, or that the number of modes can be very large. Intuitively, the availability of many operation modes gives fine control on how to operate a PMC in such a way that power waste is minimized and performance is perfectly calibrated on the task. In practice, the number of modes of operation tends to be quite small because the increased design complexity and hardware overhead for supporting power management must be tightly controlled. Several implementation techniques for PMC's are surveyed in Section IV. Here, we just stress the fact that the increased flexibility offered by PMC's may have a cost that should be taken into account.

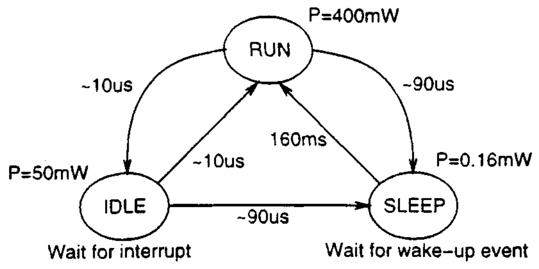


Fig. 1. Power state machine for the StrongARM SA-1100 processor.

Another important characteristic of real-life PMC's is that transitions between modes of operation have a cost. In many cases, the cost is in terms of delay, or performance loss. If a transition is not instantaneous, and the component is not operational during a transition, performance is lost whenever a transition is initiated. Transition cost depends on PMC implementation: in some cases (see Section IV) the cost may be negligible, but, generally, it is not. There might also be a transition power cost: this is often the case when transitions are not instantaneous. It is important not to neglect transition costs when designing a PMC's. Excessive costs may make one or more low-power operation states almost useless because it is very hard to amortize the cost of transitioning in and out of them.

In most practical instances, we can model a PMC by a finite-state representation called *power state machine* (PSM). States are the various modes of operation that span the tradeoff between performance and power. State transitions have a power and delay cost. In general, low-power states have lower performance and larger transition latency than states with higher power. This simple abstract model holds for many single-chip components like processors [14] and memories [7] as well as for devices such as disk drives [18], wireless network interfaces [19], displays [18], which are more heterogeneous and complex than a single chip.

Example 2.1: The StrongARM SA-1100 processor [3] is an example of PMC. It has three modes of operation: `Run`, `IDLE`, and `SLEEP`. `Run` mode is the normal operating mode of the SA-1100: every on-chip resource is functional. The chip enters run mode after successful power-up and reset. `IDLE` mode allows a software application to stop the CPU when not in use, while continuing to monitor interrupt requests on or off chip. In idle mode, the CPU can be brought back to run mode quickly when an interrupt occurs. `SLEEP` mode offers the greatest power savings and consequently the lowest level of available functionality. In the transition from `Run` or `IDLE`, the SA-1100 performs an orderly shutdown of on-chip activity. In a transition from `SLEEP` to any other state, the chip steps through a rather complex wake-up sequence before it can resume normal activity.

The PSM model of the StrongARM SA-1100 is shown in Fig. 1. States are marked with power dissipation and performance values, edges are marked with transition times. The power consumed during transitions is approximatively equal to that in `Run` mode. Notice that both `Idle` and `SLEEP` have null performance, but the time for exiting `SLEEP` is much longer than that for exiting `Idle` (10 μs versus 160 ms). On the other hand, the power consumed by the chip in `SLEEP` mode (0.16 mW) is much smaller than that in `Idle` (50 mW). □

Power-manageable components can be managed *internally* or *externally*, according to the physical location of the implementation of the corresponding policy. Internally managed components (also called *self-managed* components) use conservative policies because of the lack of observability of the overall system operation and of the need of tolerating little or no performance degradation, since no assumptions can be made on how demanding the component's environment will be. Nevertheless, there are several examples of components that are either partially of completely self-managed.

Example 2.2: IBM's Travelstar [4] hard disk drives have three low-power inactive states called `Performance Idle`, `Active Idle`, and `Low Power Idle`. When the disk is idle, the drives employ a proprietary internal management technology called "Enhanced Adaptive Battery Life Extender" for selecting the appropriate idle mode to minimize power usage. Idle-mode selection is based on the current disk drive access patterns, and IBM claims that automatic adaptation helps in improving access times. The disk does not need an external power manager and no configuration or set up is needed. Travelstar drives have also two additional very low-power states, namely, `Stand-by` and `Sleep`. The times required for entering and exiting these two states are much longer than those needed for transitioning to and from the first three. Decisions on transitions to `Stand-by` or `Sleep` are left to external control. □

B. Power-Managed Systems

From our viewpoint, a system is a set of interacting components, some of which (at least one) are externally controllable PMC's. Notice that this generic definition does not pose any limitation on the size and complexity of a system. The activity of components is coordinated by a system controller. In complex systems, control is often implemented in software. For instance, in computer systems, global coordination is performed by the *operating system* (OS).

The system controller has precise and up-to-date control on the status of system components, hence, the power manager is naturally implemented as a module of the system controller. A power-manageable system should provide a clean abstraction of its components to the power manager. Standardization of the interface between PM and system is an important feature for decreasing design time.

The choice and realization of a DPM scheme requires modeling both the components' power/performance behavior and their workload. The former model is captured well by the power state machine model. On the other hand, models for the workload may vary in complexity, and range from the simple assumption used in timeout schemes to complex statistical models. As we shall see in Section III, workload information is required for all advanced power management approaches. Hence, we postulate the existence of a system-monitoring module which is capable of collecting run-time workload data and extracting the relevant information required to drive the PM. The abstract structure of a generic system-level PM is shown in Fig. 2. The

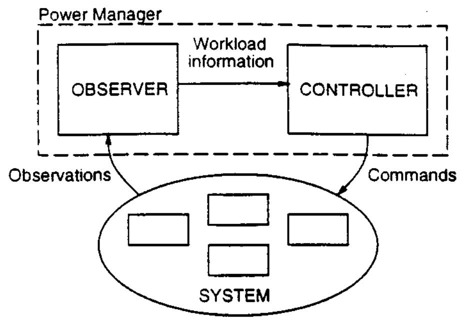


Fig. 2. Abstract structure of a system-level power manager.

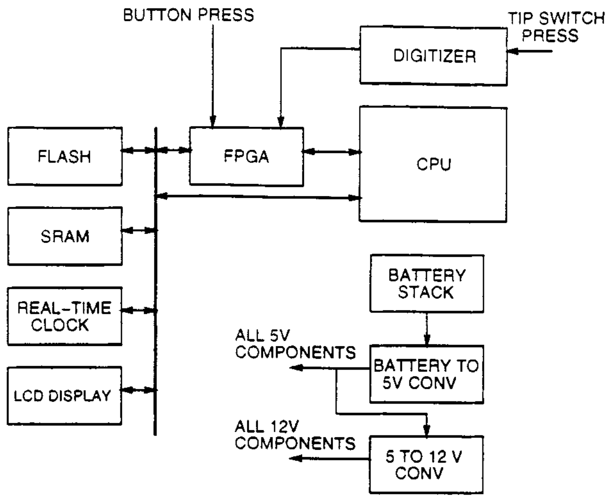


Fig. 3. PaperClip hardware diagram.

observer block collects workload information for all PMC's in the system, while the *controller* takes care of issuing commands for forcing state transitions.

Not all components in a power-managed system have to be PMC's. The power consumption of all noncontrollable components makes up a baseline power consumption that cannot be reduced by power management. Self-managed components appear as noncontrollable to the PM. Even though the functionality of the PM is clearly defined, its implementation is not constrained in any way. In some systems, the PM is a hardware block, while in others it is a software routine. Hybrid hardware–software implementations are also possible. PM implementation issues are analyzed in Section IV.

Example 2.3: The PaperClip, a hand-held electronic clipboard developed by HP Laboratories, is an example of a power-managed system [5]. The high-level hardware organization of PaperClip is shown in Fig. 3. All major components are power-manageable: the CPU, memory, LCD, and digitizer can be put in a low-power sleep state. Some components, like the real-time clock and the FPGA-based control logic, are always active. PaperClip's inputs come either from the control buttons situated on the clipboard or from the digitizer's pen. PaperClip can operate as a digitizer and as data-transfer unit. During digitize, PaperClip stores digitized handwriting on FLASH memory, while the user writes on a sheet of paper on the clipboard. During data transfer, the digitized handwriting is transferred to a host PC via either a serial or an IR interface.

Power management for PaperClip is based on a hybrid hardware–software implementation. The core PM functionality is implemented as firmware running on the CPU. PaperClip's workload can be widely varying over time. If the user is not writing on the clipboard, the system is idle. However, PaperClip should not be turned off as soon as writing stops because resuming normal operation after a sleep period requires a few milliseconds. If the PM puts the system to sleep too greedily, a significant amount of data can be lost when writing resumes, and the quality of handwriting digitization may be compromised. Power state transitions for non-CPU components are forced by the PM module running on the CPU by writing to memory-mapped I/O locations. PM commands are then decoded by control circuitry implemented with an FPGA and distributed to the components. CPU shutdown is software based. Wake-up is interrupt driven: interrupts are generated by pressing control buttons or by pressing the pen on the clipboard. Notice that sleep power cannot be reduced to zero because some of the system components are not power manageable. □

C. Power-Managed Networks

In many cases, systems are not isolated, but they actively communicate among themselves. We call *network* a set of communicating systems. While network design has been traditionally focused on communication quality and throughput, the increased emphasis on low-power portable systems with communication capabilities has spurred several research initiatives targeting power-efficient networking [22].

Energy-conscious communication protocols based on power management have been extensively studied [23]–[25]. 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. Even when interference is not an issue, point-to-point communication can be made more power efficient by increasing the predictability of communication patterns [26]: if it is possible to accurately predict the arrival time of messages (packets), idle times can be exploited to force communication devices into a low-power inactive state.

The main challenge in network power management is that it is generally not realistic to assume that power control is centralized. Hence, we must rely on distributed algorithms that take autonomous decisions for each system in the network based either on local information, or on incomplete global network status data. Even though network power management is an interesting and relevant topic, we focus on system-level centralized power management.

III. Dynamic Power Management Techniques

In this section, we analyze techniques for controlling the power state of a system and its components. We consider components as black boxes, whose behavior is abstracted by the PSM model. We defer to Section IV the description of the interfacing layers for component control as well as the implementation technical details. We focus on how to design effective power management policies. For the sake of simplicity, we shall focus on the problem of controlling a single component (or, equivalently, the system as a whole).

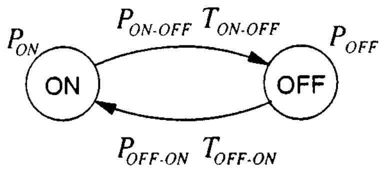

P_{ON}
ON
$P_{ON\text{-}OFF}$ $T_{ON\text{-}OFF}$
P_{OFF}
OFF
$P_{OFF\text{-}ON}$ $T_{OFF\text{-}ON}$

Fig. 4. PSM of a two-state power-manageable component.

First, we want to clarify why the search for a DPM policy is not a simple problem to solve. For this reason, we give an example of a trivial problem first. Consider a system where transitions between power states are instantaneous: negligible power and performance costs are paid for performing state transitions. In such a system, DPM is a trivial task, and the optimum policy is greedy: as soon as the system is idle, it can be transitioned to the deepest sleep state available. On the arrival of a request, the system is instantaneously activated.

Unfortunately, most PMC's have nonnegligible performance and power costs for power state transitions. For instance, if entering a low-power state requires power-supply shutdown, returning from this state to the active state requires a (possibly long) time for: 1) turning on and stabilizing the power supply and the clock; 2) reinitializing the system; and 3) restoring the context. When power state transitions have a cost, as it is typically the case, we are faced with a difficult optimization problem. In rough but intuitive words, we need to decide when (if at all) it is worthwhile (performance and power-wise) to transition to a low-power state and which state should be chosen (if multiple low-power states are available).

Example 3.1: Consider the StrongARM SA-1100 processor described in Example 2.1. Transition times between `Run` and `Idle` states are so fast that the `Idle` state can be optimally exploited according to a greedy policy possibly implemented by an embedded PM.

On the other hand, the wake-up time from the `Sleep` state is much larger and has to be carefully compared with the environment's time constants before deciding to shut the processor down. In the limiting case of a workload with no idle periods longer than the time required to enter and exit the `Sleep` state, a greedy policy shutting down the processor as soon as an idle period is detected would reduce performance without saving any power (the power consumption associated with state transitions is of the same order of that of the `Run` state). An external PM controlling transitions of the SA-1100 processor to the `Sleep` state has to observe the workload and take decisions according to a policy whose optimality depends on workload statistics and on predefined performance constraints. Notice that the policy becomes trivial if there are no performance constraints: the PM could keep the processor always in the `Sleep` state.

An SA-1100 processor with embedded control for the `Idle` state and external control for the `Sleep` state is a partially self-managed PMC whose PSM model (shown in Fig. 4) has only two states: `On` and `Off`. The `On` state is a macrostate representing both the `Run` and `Idle` states of the processor, with a greedy policy autonomously controlling transitions between them. The power consumption associated with the `On` state is the sum of the power consumptions of the `Run` and `Idle` states, weighted by the workload activity and idleness probabilities. The `Off` state corresponds to the actual `Sleep` state. Transitions between `On` and `Off` represent transitions between the `Run` and `Sleep` states. □

Example 3.1 leads to two observations. First, policy optimization is a *power optimization* problem under *performance constraints*, or vice versa. Second, the achievable power savings depend on the workload (which must be bursty at some degree), and system characteristics (i.e., the PSM of the system). The general applicability of DPM is discussed in the next section as a property of a system-workload pair. Existing techniques for DPM and policy optimization are surveyed and discussed in Sections III-B and C, focusing on *predictive techniques* and *stochastic control*, respectively.

A. Applicability of DPM

Putting a PMC into an inactive state causes a period of inactivity whose duration T_n is the sum of the actual time spent in the target state and the time spent to enter and exit it. We define the *break-even time* for an inactive state S (denoted by $T_{BE,S}$) as the minimum inactivity time required to compensate the cost of entering state S. The break-even time $T_{BE,S}$ is inferred directly from the power state machine of a PMC. If $T_n < T_{BE,S}$, either there is not enough time to enter and exit the inactive state, or the power saved when in the inactive state does not amortize the additional power consumption typically required to turn-on the component. Intuitively, DPM aims at exploiting idleness to transition a component to an inactive low-power state. If no performance loss is tolerated, the length of the idle periods of the workload is an upper bound for the inactivity time of the resource. On the other hand, if some performance loss is tolerated, inactivity times may be longer than idle periods.

In this section, we analyze the *exploitability* of the inactive states of a PMC, that is the possibility of saving power by transitioning the component to the inactive states. Exploitability depends on the power states, on the workload, on the performance constraints, on the DPM policy and on the PM implementation. Techniques for policy optimization and implementation will be discussed later, together with the impact of performance constraints. Here, we focus only on *inherent exploitability*, which represents the possibility of exploiting an inactive state under the assumption that: 1) no performance penalty is tolerated and 2) an *ideal PM* is available that has complete (*a priori*) knowledge of the entire workload trace. Inherent exploitability is a property of a system-workload pair.

For example, consider the two-state PSM of a component, as shown in Fig. 4. For the sake of clarity, when there is only one inactive state, we will use the shorthand notation T_{BE} instead of $T_{BE,S}$. The optimum policy for an ideal PM controlling the transitions between states `On` and `Off` consists of shutting down the component at the beginning of all idle periods longer than $T_{BE} = T_{BE,\text{Off}}$ and waking it up right in time to serve upcoming requests with no delay. The resulting power consumption (denoted by P_{ideal}) is a lower bound for the power consumption that can be achieved by a PM exploiting inactive state `Off`. The potential power saving (P_{saved}), defined as the gap between P_{ideal} and the power consumption of the system when in the active state (P_{On}), represents the inherent exploitability of the `Off` state for the given workload. The larger P_{saved} the larger the potential advantage of exploiting state `Off` for DPM. If $P_{\text{saved}} = 0$, no power savings can be achieved by entering the inactive state without impairing performance. Needless to say, the inactive state can always be exploited in practice if arbitrary performance degradation is tolerated.

We are interested in studying the dependence of P_{saved} on power-state parameters and workload statistics. The parameters of a power state S are represented by its break-even time $T_{BE,S}$, while workload statistics are represented by the probability distribution of the idle periods $F(T_{\text{idle}})$. Intuitively, the larger $T_{BE,S}$ (with respect to the average idle time), the smaller P_{saved}. In the limiting situation where all idle periods are shorter than $T_{BE,S}$, no power savings would be achieved by means of DPM: an ideal PM implementing the optimum policy would never shut the resource down, thus providing $P_{\text{ideal}} = P_{\text{On}}$ and $P_{\text{saved}} = 0$.

In general, T_{BE} is the sum of two terms: the total transition time (i.e., the time required to enter and exit the inactive state, T_{TR}) and the minimum time that has to be spent in the low-power state to compensate the additional transition power (P_{TR}). For our example PMC (Fig. 4), T_{TR} and P_{TR} can be computed as

$$T_{TR} = T_{\text{On, Off}} + T_{\text{Off, On}} \tag{1}$$

$$P_{TR} = \frac{T_{\text{On, Off}}P_{\text{On, Off}} + T_{\text{Off, On}}P_{\text{Off, On}}}{T_{TR}} \tag{2}$$

while T_{BE} can be expressed as

$$\begin{aligned} T_{BE} &= T_{TR} + T_{TR}\frac{P_{TR} - P_{\text{On}}}{P_{\text{On}} - P_{\text{Off}}} \text{ if } P_{TR} > P_{\text{On}} \\ T_{BE} &= T_{TR} \text{ if } P_{TR} \le P_{\text{On}}. \end{aligned} \tag{3}$$

In practice, T_{BE} grows linearly with transition time and cost (T_{TR} and P_{TR}) and depends hyperbolically on the power saved ($P_{\text{On}} - P_{\text{Off}}$) when in the inactive state When $P_{TR} \le P_{\text{On}}$, T_{BE} reduces to T_{TR} (this is, for instance, the case of the ARM SA-1100 processor), while it is greater than T_{TR} when $P_{TR} > P_{\text{On}}$ (as for components with mechanical inertia, such as hard disk drives). In this case, we need to add to T_{TR} the term $T_{TR}(P_{TR} - P_{\text{On}})/(P_{\text{On}} - P_{\text{Off}})$, which represents the additional time that we need to spend in the `Off` state to compensate the excess power consumed during state transition.

For systems with multiple inactive states, a different break-even time $T_{BE,S}$ and, consequently, a different value of $P_{\text{saved},S}$, has to be defined for each state S. Deeper sleep states have lower power consumption at the cost of longer and more expensive transitions. When designing power-manageable components, a tradeoff between P_S, P_{TR} and T_{TR} has to be found for each sleep state S to obtain small values of $T_{BE,S}$ and high exploitability. Sleep states with smaller break-even times are more likely to be successfully exploited by DPM.

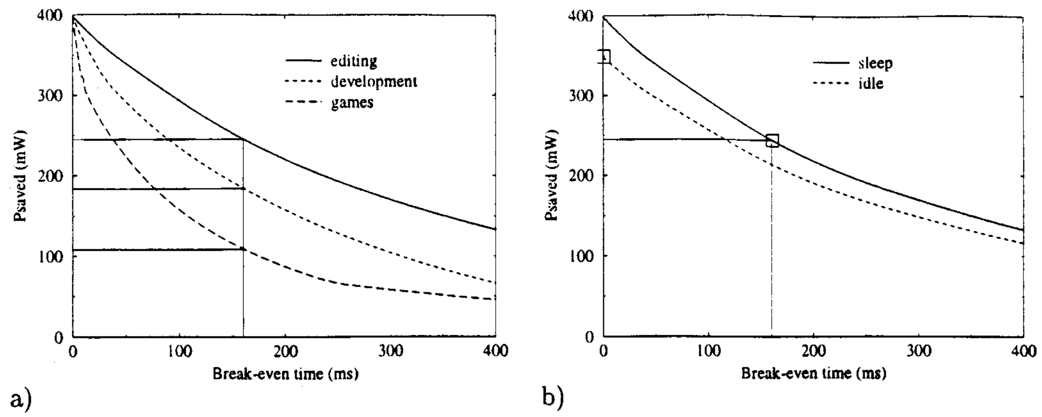


Fig. 5. (a) Plot of $P_{\text{saved}}(T_{BE})$ for the `Sleep` state of the StrongARM SA-1100 processor. The three curves refer to three different workload statistics, computed from real-world CPU traces provided by the *IPM monitoring package* [5]. (b) Comparison of $P_{\text{saved}}(T_{BE})$ for the two inactive states of the SA-1100 processor. The two curves refer to the same workload.

The energy saved by entering state S during an idle period $T_{\text{idle}} > T_{BE,\,S}$ is

$$E_S(T_{\text{idle}}) = (T_{\text{idle}} - T_{TR})(P_{\text{On}} - P_S) + T_{TR}(P_{\text{On}} - P_{TR}). \tag{4}$$

Its average value is given by

$$E_S^{\text{avg}} = \int_{T_{BE}}^{\infty} E_S(T_{\text{idle}}) f(T_{\text{idle}})\, dT_{\text{idle}} \tag{5}$$

where $f(T_{\text{idle}})$ is the probability density of the idle periods. The exploitability of S (in symbols, $P_{\text{saved},\,S}$) is the ratio between E_S^{avg} and the average length of the idle periods ($T_{\text{idle}}^{\text{avg}}$). By replacing the expression of $E_S(T_{\text{idle}})$ from (4) into (5) and dividing by $T_{\text{idle}}^{\text{avg}}$, we obtain a formula for $P_{\text{saved},\,S}$

$$P_{\text{saved},\,S} = \frac{1}{T_{\text{idle}}^{\text{avg}}} \int_{T_{BE}}^{\infty} [(T_{\text{idle}} - T_{TR})(P_{\text{On}} - P_S) + T_{TR}(P_{\text{On}} - P_{TR})] f(T_{\text{idle}})\, dT_{\text{idle}} \tag{6}$$

which can be integrated and rewritten as the product of three terms: the power saving of state S, the expected idle time in excess of $T_{BE,\,S}$ (normalized at the average idle period), and the probability of going to state S (assuming that we perform the transition only when it is convenient)

$$P_{\text{saved},\,S} = (P_{\text{On}} - P_S) \frac{\left(T_{\text{idle}>T_{BE,\,S}}^{\text{avg}} - T_{BE,\,S}\right)}{T_{\text{idle}}^{\text{avg}}} \cdot (1 - F(T_{BE})) \tag{7}$$

where F is the probability distribution of T_{idle} and $T_{\text{idle}>T_{BE,\,S}}^{\text{avg}}$ is the average length of idle periods longer then $T_{BE,\,S}$. The power saved P_{saved} is always a decreasing function of $T_{BE,\,S}$: it takes maximum value for $T_{BE,\,S} = 0$ and asymptotically tends to zero for increasing values of $T_{BE,\,S}$. The way it goes to zero depends on the first-order statistics of the workload, namely, on the distribution of T_{idle}.

Example 3.2: We want to evaluate the exploitability of the inactive states of the StrongARM SA-1100 processor. We start by computing their break-even times according to (4). Since the power consumption associated with all state transitions is equal to P_{Run}, $T_{BE} = T_{TR}$

$$T_{BE,\,\text{Idle}} = 0.01 \text{ ms} + 0.01 \text{ ms}$$
$$T_{BE,\,\text{Sleep}} = 160 \text{ ms} + 0.09 \text{ ms}.$$

As intuitively observed at the beginning of this section, the `Idle` state has a break-even time much smaller than the `Sleep` state.

As reference workloads to evaluate exploitability, we take real-world CPU usage traces provided by the IPM monitoring system [5] described in Section IV. From each trace, we compute the probability distribution function $F(T_{\text{idle}})$ and we evaluate (7) for different values of T_{BE}. The behavior of $P_{\text{saved, Sleep}}$ as a function of the break-even time is shown in Fig. 5(a) for three different CPU workloads, corresponding to three different user sessions: editing, software development, and graphical interactive games. The dependence on the workload is evident: graphical interactive games require more CPU usage than text editors, thus reducing the opportunity of putting the CPU to the `Sleep` state. Notice that, if the break-even time for the `Sleep` state were null, $P_{\text{saved, Sleep}}$ would have been of about 400 mW independently of the workload. Corresponding to the actual value of $T_{BE,\,\text{Sleep}}$, instead, $P_{\text{saved, Sleep}}$ is much smaller and strongly dependent on the workload.

Fig. 5(b) compares the P_{saved} curves of both inactive states for the same workload (namely, the editing trace). $P_{\text{saved},\,Idle}$ is always below $P_{\text{saved, Sleep}}$. Since the `Sleep` state has lower power consumption than the `Idle` state, if the two states had the same break-even time the deepest one would have been more exploitable. However, taking into account the actual break-even times we find that the inherent exploitability of the `Idle` state is greater than that of the `Sleep` state (the points to be compared are shown by square boxes on the graph). □

As mentioned at the beginning of the section and formally expressed by (7), the exploitability of an inactive state depends both on the characteristics of the inactive state and on the workload. If typical workload information is not available when designing a PMC, the exploitability of its low-power states cannot be computed. To represent the properties of an inactive state S independently of the workload, we use the time-power product

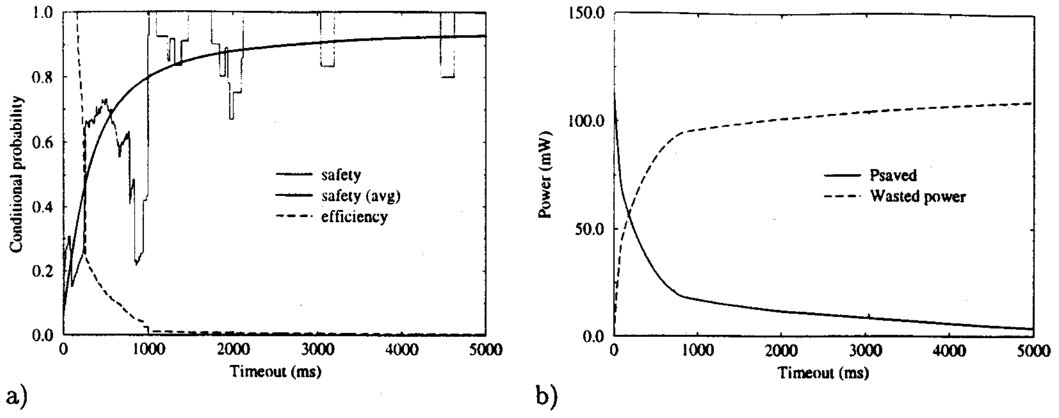


Fig. 6. Quality of a timeout-based predictor evaluated as a function of timer duration. (a) Safety and efficiency of the timeout used to predict idle periods longer than $T_{BE} = 160$ ms. (b) Saved and wasted power consumption. Data refer to the PSM of Example 3.1 and to a CPU usage trace provided by the IPM monitoring package [5].

$C_S = T_{BE,S} \cdot P_S$. Inactive states with lower C_S are likely to lead to larger power savings. Incidentally, we remark that C_S has the same dimension of the well-known power-delay product used as a cost metric for comparing different electronic devices and circuits.

B. Predictive Techniques

In most real-world systems, there is little knowledge of future input events and DPM decisions have to be taken based on uncertain predictions. The rationale in all predictive techniques is that of exploiting the correlation between the past history of the workload and its near future in order to make reliable predictions about future events. We denote by p the future event that we want to predict. We denote by o the past event whose occurrence is used to make predictions on p. For the purpose of DPM we are interested in predicting idle periods long enough to go to sleep, in symbols: $p = \{T_{\text{idle}} > T_{BE}\}$.

Good predictors should minimize the number of mispredictions. We call *overprediction* (*underprediction*) a predicted idle period longer (shorter) than the actual one. Overpredictions give rise to a performance penalty, while underpredictions imply power waste but no performance penalty. To represent the quality of a predictor we define two figures: *safety*, that is the complement of the risk of making overpredictions, and *efficiency*, that is the complement of the risk of making underpredictions. Safety and efficiency can be expressed in terms of conditional probabilities $\text{Prob}(p|o)$ and $\text{Prob}(o|p)$. A totally safe predictor never makes overpredictions ($\text{Prob}(p|o) = 1$), and a totally efficient predictor never makes underpredictions ($\text{Prob}(o|p) = 1$). A predictor with maximum safety and efficiency is an *ideal predictor*, whose availability would enable the actual implementation of the ideal PM discussed in the previous section. Predictors of practical interest are neither safe nor efficient, thus causing suboptimum control. Their quality (and the quality of the resulting control) depends on the choice of the observed event o and on the second-order workload statistics.

1) Static Techniques:

Fixed Timeout: The most common predictive PM policy is the *fixed timeout*, which uses the elapsed idle time as observed event ($o = \{T_{\text{idle}} > T_{TO}\}$) to be used to predict the total duration of the current idle period ($p = \{T_{\text{idle}} > T_{TO} + T_{BE}\}$). The policy can be summarized as follows: when an idle period begins, a timer is started with duration T_{TO}. If after T_{TO} the system is still idle, then the PM forces the transition to the `Off` state. The system remains off until it receives a request from the environment that signals the end of the idle period. The fundamental assumption in the fixed timeout policy is that the probability of T_{idle} being longer than $T_{BE}+T_{TO}$, given that $T_{\text{idle}} > T_{TO}$, is close to one: $\text{Prob}(T_{\text{idle}} > T_{TO}+T_{BE}|T_{\text{idle}} > T_{TO}) \approx 1$. The critical design decision is obviously the choice of the timeout value T_{TO}.

Timeouts have two main advantages: they are general (their applicability slightly depends on the workload) and their safety can be improved simply by increasing the timeout values. Unfortunately, they tradeoff efficiency for safety: large timeouts cause a large number of underpredictions, which represent a missed opportunity of saving power, and a sizeable amount of power is wasted waiting for the timeout to expire.

Example 3.3: Consider one of the CPU usage traces described in Example 3.2 (namely, the game trace) as a typical workload for the StrongARM SA-1100 processor. We want to evaluate the quality of a timeout-based shutdown policy for the processor. Since the break-even time for the `Sleep` state is of 160 ms, we evaluate the safety and efficiency of a timeout used to predict idle periods longer than 160 ms. The two figures are plotted on Fig. 6(a) as a function of the timer duration. As the timeout increases, predictions become safer but less efficient (efficiency is almost null for timeouts greater than 1 s). It is also worth noting that safety has a highly nonsmooth instance-dependent behavior that makes it difficult to choose optimal timeout values [the irregular curve in Fig. 6(a) refers to a 1-h trace, while the smooth one refers to the average of several traces collected during equivalent user sessions].

Fig. 6(b) shows the power savings obtained by applying the timeout policy to the SA-1100 and the wasted power evaluated with respect to the ideal power savings. The effect of T_{TO} on the actual power savings is similar to the effect of T_{BE} on the ideal ones. Both parameters reduce the portion of idle time that can be effectively exploited to save power. □

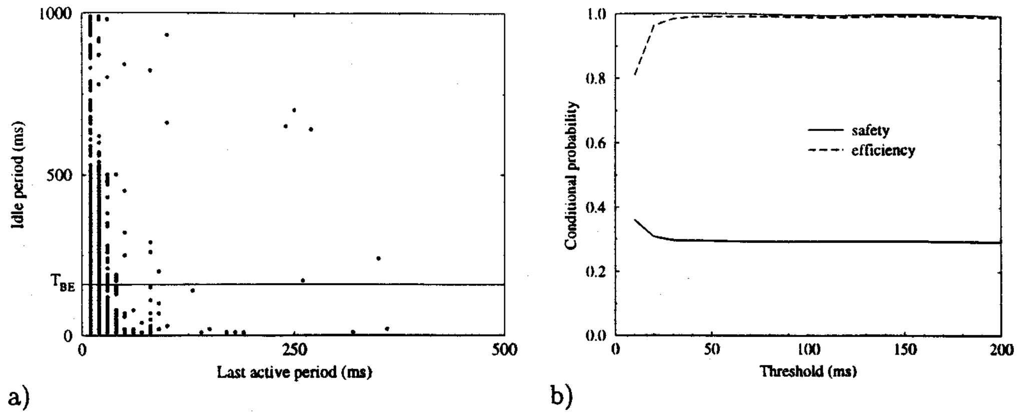


Fig. 7. (a) Scatter plot of T_{idle} versus T_{active} for the workload of the CPU of a personal computer running Linux. (b) Safety and efficiency of a predictive shutdown scheme plotted as a function of the threshold value T_{Thr}.

Karlin *et al.* [31] proposed to use $T_{TO} = T_{BE}$ and showed that this choice leads to an energy consumption which is at worse twice the energy consumed by an ideal policy. The rationale of this strong result is related to the fact that the worst case happens for traces with repeated idle periods of length $T_{\text{idle}} = 2T_{BE}$ separated by pointwise activity. In this case, Karlin's algorithm provides no power saving, while an ideal algorithm saves power during half of each idle interval. Indeed, the ideal algorithm performs a shutdown for each idle period, but half of the period is spent in state transition.

Timeout schemes have two more limitations: they waste a sizeable amount of power (during user's idleness) waiting for the timeout to expire and they always pay a performance penalty upon wakeup. The first issue is addressed by *predictive shutdown policies* [30], [32] that take PM decisions as soon as a new idle period starts, based on the observation of past idle and busy periods. The second issue is addressed by predictive wakeup, described later.

Predictive Shutdown: Two predictive shutdown schemes have been proposed by Srivastava *et al.* [32]. In the first scheme, a nonlinear regression equation is obtained from the past history

$$T_{\text{pred}} = \phi\left(T^{n}_{\text{active}}, T^{n-1}_{\text{idle}}, \cdots, T^{n-k}_{\text{active}}, T^{n-k-1}_{\text{idle}}\right) \tag{8}$$

and used to make predictions. We use superscripts to indicate the sequence of past idle and active periods; n indicates the current idle period (whose length has to be predicted) and the most recent active period. If $T_{\text{pred}} > T_{BE}$, the system is immediately shut down as soon as it becomes idle. According to our notation, the observed event is

$$o = \left\{\phi\left(T^{n}_{\text{active}}, T^{n-1}_{\text{idle}}, \cdots, T^{n-k}_{\text{active}}, T^{n-k-1}_{\text{idle}}\right) > T_{BE}\right\}. \tag{9}$$

The format of the nonlinear regression is decided heuristically, while the fitting coefficients can be computed with standard techniques. The main limitations of this approach are: 1) there is no automatic way to decide the type of regression equation and 2) offline data collection and analysis are required to construct and fit the regression model.

The second approach proposed by Srivastava *et al.* [32] is based on a *threshold*. The duration of the busy period immediately preceding the current idle period is observed. If $o = \{T^{n-1}_{\text{active}} < T_{Thr}\}$, the idle period is assumed to be larger than T_{BE} and the system is shut down. The rationale of this policy is that for the class of systems considered by Srivastava *et al.*(interactive graphic terminals), short active periods are often followed by long idle periods. Clearly, the choice of T_{Thr} is critical. Careful analysis of the scatter plot of T_{idle} versus T_{active} is required to set it to a correct value, hence, this method is inherently offline (i.e., based on extensive data collection and analysis). Furthermore, the method is not applicable if the scatter plot is not L-shaped.

Example 3.4: Fig. 7(a) shows the scatter plot of T_{idle} versus T_{active} for the development trace of Example 3.2. From the plot, we observe that: 1) the time is discretized (both T_{idle} and T_{active} are multiple of 10 ms, that is the duration of the time slots assigned by the Linux scheduler to the active process); 2) the large majority of the idle periods are shorter than 1000 ms (this is due to the presence of a system daemon that required the CPU at every second independently of the state of user's application); and 3) the scatter plot is L-shaped (thus enabling the use of threshold-based predictors). The horizontal line shows the break-even time of the sleep state of the StrongARM SA-1100 processor. Safety and efficiency of a threshold-based predictor used to shut down the SA-1100 are plotted in Fig. 7(b) as a function of T_{Thr}. Interestingly, efficiency becomes almost one even for small threshold values (in fact, most of the exploitable idle periods are preceded by short active periods), but there is no way of improving safety. In our example, threshold-based predictions are unsafe due to the presence of a dense region in the bottom-left corner of the scatter plot. A threshold on T_{active} does not help us in distinguishing between idle periods longer or shorter than T_{BE}. $\square$

The applicability and the quality of history-based predictors depend on the correlation between past and future events, that is, not under designer's control. As a matter of fact, short-term correlation has been observed in many real-world workloads, but the nature and strength of such correlation is strongly instance dependent. For a given workload, history-based predictors are usually more efficient and less safe than timeouts.

Predictive Wakeup: The DPM strategy proposed by Hwang *et al.* [33] addresses the second limitation of timeout policies, namely the performance penalty that is always paid on wakeup.

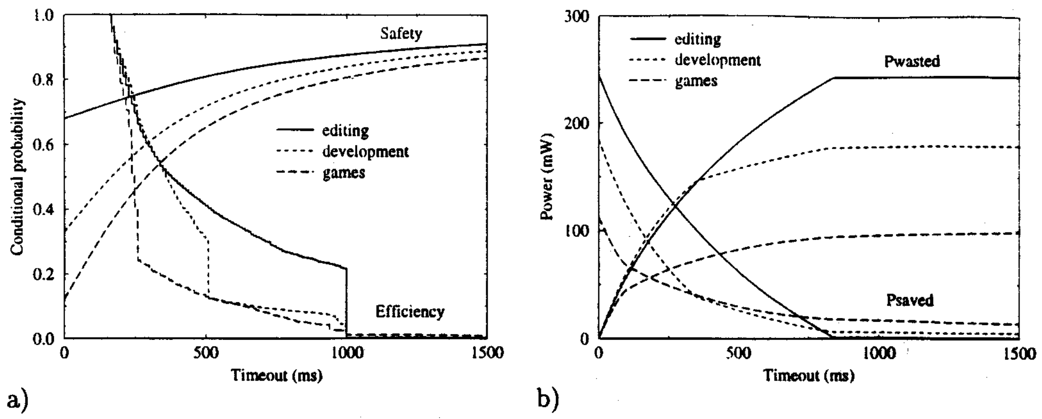


Fig. 8. Effect of the workload on the quality of a timeout-based power manager. (a) Safety and efficiency. (b) Saved and wasted power.

To reduce this cost, the power manager performs *predictive wakeup* when the predicted idle time expires, even if no new requests have arrived. This choice may increase power dissipation if $T_{\rm idle}$ has been underpredicted, but decreases the delay for servicing the first incoming request after an idle period.

2) Adaptive Techniques: Since the optimality of DPM strategies depends on the workload statistics, static predictive techniques are all ineffective (i.e., suboptimal) when the workload is either unknown *a priori*, or nonstationary. Hence, some form of adaptation is required. While for timeouts the only parameter to be adjusted is the timer duration, for history-based predictors even the type of observed events could in principle be adapted to the workload.

Example 3.5: Fig. 8 shows the same graphs of Fig. 6 plotted for three different workloads. All the parameters used in Example 3.3 to represent the quality of a timeout-based estimator are shown to be strongly dependent on the workload. Suppose, for instance, that a target power saving (e.g., of 50 mW) has to be guaranteed regardless of the performance degradation. For a given workload (namely, the editing trace) the timeout value to be used to meet the constraint can be obtained from the corresponding curve of Fig. 8(b): about 550 ms. However, as the workload changes (becoming for instance similar to the development trace), the fixed timeout does not guarantee the required power savings any longer (for the development trace, the power savings provided by a timeout of 550 ms are of about 25 mW). □

Several adaptive predictive techniques have been proposed to deal with nonstationary workloads. In the work by Krishnan *et al.* [27], a set of timeout values is maintained and each timeout is associated with an index indicating how successful it would have been. The policy chooses, at each idle time, the timeout that would have performed best among the set of available ones. Another policy, presented by Helmbold *et al.* [28], also keeps a list of candidate timeouts and assigns a weight to each timeout based on how well it would have performed relatively to an optimum offline strategy for past requests. The actual timeout is obtained as a weighted average of all candidates with their weights. Another approach, introduced by Douglis *et al.* [29], is to keep only one timeout value and to increase it when it is causing too many shutdowns. The timeout is decreased when more shutdowns can be tolerated. Several predictive policies are surveyed and classified in Douglis' paper.

Another aggressive shutdown policy has been proposed by Hwang *et al.* [33]. This policy is capable of online adaptation, since the predicted idle time $T^n_{\rm pred}$ is obtained as a weighted sum of the last idle period $T^{n-1}_{\rm idle}$ and the last prediction $T^{n-1}_{\rm pred}$

$$T^n_{\rm pred} = aT^{n-1}_{\rm idle} + (1-a)T^{n-1}_{\rm pred}. \tag{10}$$

This recursive formula dynamically changes the actual observed event: $o = \{aT^{n-1}_{\rm idle} + (1-a)T^{n-1}_{\rm pred} > T_{BE}\}$.

Underprediction impact is mitigated by employing a timeout scheme to reevaluate $T_{\rm pred}$ periodically if the system is idle and it has not been shut down. Overprediction impact is reduced by imposing a saturation condition on predictions: $T^n_{\rm pred} < C_{\rm max}T^{n-1}_{\rm pred}$.

Workload prediction accuracy can be increased by specializing predictors to particular classes of workload. Specialization restricts the scope of applicability, but it also reduces the difficulties of predicting completely general workloads. A recently proposed adaptive technique [34] is specifically tailored toward hard-disk power management and it is based on the observation that disk accesses are clustered in *sessions*. Sessions are periods of relatively high disk activity separated by long periods of inactivity. Under the assumption that disk accesses are clustered in sessions, adaptation is used only to predict *session length*. Prediction of a single parameter is easily accomplished and the reported accuracy is high.

C. Stochastic Control

Policy optimization is an optimization problem under uncertainty. Predictive approaches address workload uncertainty, but they assume deterministic response and transition times for the system. However, the system model for policy optimization is very abstract, and abstraction introduces uncertainty. Hence, it may be safer, and more general, to assume a stochastic model for the system as well. Moreover, predictive algorithms are based on a two-state system model, while real-life systems have multiple power states. Policy optimization involves not only the choice of *when* to perform state transitions, but also the choice of *which* transition should be performed. Furthermore, predictive algorithms are heuristic, and their optimality can only be gauged

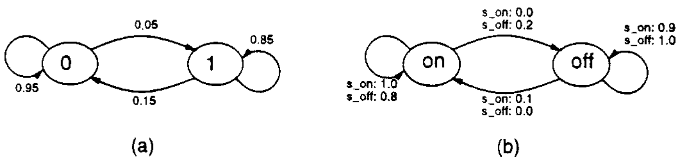


Fig. 9. Markov model of a power-managed system and its environment.

through comparative simulation. Parameter tuning for these algorithms can be very hard if many parameters are involved. Finally, predictive algorithms are geared toward power minimization, and cannot finely control performance penalty.

The stochastic control approach addresses the generality and optimality issues outlined above. Rather than trying to eliminate uncertainty by prediction, it formulates policy optimization as an optimization problem under uncertainty. More specifically [39], power management optimization has been studied within the framework of *controlled Markov processes* [42], [43]. In this flavor of stochastic optimization, it is assumed that the system and the workload can be modeled as Markov chains. Under this assumption, it is possible to: 1) model the uncertainty in system power consumption and response (transition) times; 2) model complex systems with many power states, buffers, queues, etc.; 3) compute power management policies that are globally optimum; and 4) explore tradeoffs between power and performance in a controlled fashion. The Markov model postulated by the stochastic control approach [39] consists of the following.

- A *service requester* (SR), a Markov chain with state set R, which models the arrival of service requests for the system (i.e., the workload).
- A *service provider* (SP), a controlled Markov chain with S states that models the system. Its states represent the modes of operation of the system (i.e., its power states), its transitions are probabilistic, and probabilities are controlled by commands issued by the power manager.
- A *power manager* (PM), which implements a function $f\colon S \times R \rightarrow A$ from the state set of SR and SP to the set of possible commands A. Such function is an abstract representation of a decision process: the PM observes the state of the system and the workload, takes a decision, and issues a command to control the future state of the system.
- *Cost metrics*, which associate power and performance values with each system state-command pair in $S \times R \times A$.

In the work by Paleologo *et al.* [39], the general Markov model is specialized by assuming finite state set, finite command set, and discrete (or slotted) time. Continuous-time Markov models have been studied as well [37], [38], [40].

Example 3.6: A simple Markov model for a power-managed system [39] is shown in Fig. 9. The SR is a two-state Markov chain with two states: zero (no request is issued to the service provider) and one (a request is issued to the provider). The transition probabilities between states are represented as edge weights in Fig. 9(a). The chain models a "bursty" workload. There is a high probability (0.85) of receiving a request during period $n+1$ if a request was received during period n, and the mean duration of a stream of requests is equal to $1/0.15 = 6.67$ periods.

The SP model has two states as well, namely $\mathcal{S} = \{\mathtt{on}, \mathtt{off}\}$. State transitions are controlled by two commands that can be issued by the power manager. The commands are, respectively, $\mathtt{s\_on}$ and $\mathtt{s\_off}$, with the intuitive meaning of "switch on" and "switch off." When a command is issued, the SP will move to a new state in the next period with a probability dependent only on the command, and on the departure and arrival states. The Markov chain model of the SP is shown in Fig. 9(b). Edge weights represent transition probabilities. Notice that their values depend on the command issued by the power manager. A power management policy can be represented as a table that associates a command with each pair of states of SP, SR. For instance, a simple deterministic policy is: $f\colon \{(0, \mathtt{on}) \rightarrow \mathtt{s\_off}, (1, \mathtt{on}) \rightarrow \mathtt{s\_on}, (0, \mathtt{off}) \rightarrow \mathtt{s\_off}, (1, \mathtt{off}) \rightarrow \mathtt{s\_on}\}$. $\square$

1) Static Techniques: To perform policy optimization, the Markov chains of SR and SP are composed to obtain a global controlled Markov chain. Then, the problem of finding a minimum-power policy that meets given performance constraints can be cast as a linear program (LP). The solution of the LP produces a *stationary randomized* policy. Such a policy is a nondeterministic function which, given a present system state, associates a probability with each command. The command to be issued is selected by a random trial based on the state-dependent probabilities. It can be shown [43] that the policy computed by LP is *globally optimum*. Furthermore, LP can be solved in polynomial time in the number of variables. Hence, policy optimization for Markov processes is exact and computationally efficient.

Stochastic control based on Markov models has several advantages over predictive techniques. First, it captures the global view of the system, thus allowing the designer to search for a global optimum that possibly exploits multiple inactive states of multiple interacting resources. Second, it enables the exact solution (in polynomial time) of the performance-constrained power optimization problem. Third, it exploits the strength and optimality of randomized policies.

However, several important points need to be understood. First, the performance and power obtained by a policy are *expected* values, and there is no guarantee that results will be optimum for a specific workload instance (i.e., a single realization of the corresponding stochastic process). Second, policy optimization requires a Markov model for SP and SR. If we can safely assume that the SP model can be precharacterized, we cannot assume that we always know the SR model beforehand. Third, policy implementation in practice may not be straightforward. We have always implicitly assumed that the power consumption of the PM is negligible, but this assumption needs to be validated on a case-by-case basis. Finally, the Markov model for the SR or SP can be just an approximation of a much more complex stochastic process. If the model is not accurate, then the "optimal" policies are just approximate solutions.

Example 3.7: We apply stochastic control to our example system, namely, the two-state PSM of the SA-1100 processor. The only decision to be taken by the PM is when to shut down the component. We stress that this is not a typical application of stochastic control (whose main strength is the capability of managing multiple states and finding a global optimum in a

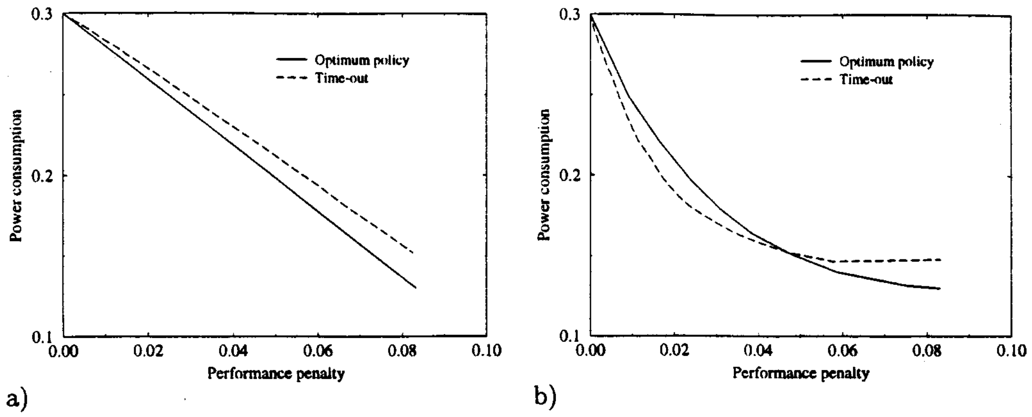


Fig. 10. Power-performance tradeoff curves for the SA-1100 with (a) a realization of a stationary Markovian workload and (b) a highly non-Markovian/nonstationary workload. Solid and dashed lines refer to stochastic control and timeout-based shutdown policies, respectively.

large design space) since there is only one sleep state and the PM cannot control the wake-up. This simple example, however, allows us to make a fair comparison between stochastic control and predictive techniques based on timeouts. The optimal Markov policy is computed by formulating a Markov chain model for the workload, composing it with the controlled Markov model extracted from the PSM of the SA-1100 and solving the LP problem associated with the controlled Markov model of processor and workload under performance constraints [39].

Comparative results for a static Markovian workload are shown in Fig. 10(a): the solid line is the performance versus power Pareto curve of optimum stochastic control (obtained by varying the performance constraint), while the dashed line is the tradeoff curve of a timeout policy (obtained by varying the timer duration). We remark that optimum stochastic control performs better than a timeout heuristic even if the degrees of freedom available for optimization are exactly the same. The difference in power is proportional to the timeout time, which represents a wasted opportunity of saving power.

The same comparison is repeated in Fig. 10 for a highly nonstationary non-Markovian workload. For several timer values, timeout-based shutdown outperforms stochastic control. In fact, policy optimization is not guaranteed to provide optimum results if the modeling assumptions are not verified. □

The class of application of stochastic control is that of computer systems subject to performance constraints. We remark, however, that policy optimization can be used as a tool for design exploration even when stochastic control is not the target DPM technique. In fact, once Markov models have been constructed for the system and the workload, the Pareto curve of optimum tradeoff points can be drawn on the power-performance plane by repeatedly solving policy optimization while varying performance constraints. The Pareto curve provides valuable information to evaluate and improve the quality of any power management strategy.

2) Adaptive Techniques: One limitation of the stochastic optimization technique described in the previous section is that it assumes complete *a priori* knowledge of the system (i.e., the SP) and its workload (SR). Even though it is generally possible to construct a model for the SP once for all, system workload is generally much harder to characterize in advance. Furthermore, workloads are often nonstationary. An adaptive extension of the static stochastic optimization approach has been presented by Chung *et al.* [41]. Adaptation is based on three simple concepts: *policy precharacterization*, *parameter learning*, and *policy interpolation*. A simple two-parameter Markov model for the workload is assumed, but the value of the two parameters is initially unknown.

Policy precharacterization constructs a two-dimensional (2-D) table addressed by values of the two parameters. The table element uniquely identified by a pair of parameters contains the optimal policy for the system under the workload uniquely identified by the pair. The table is filled by computing optimum policies under different workloads. During system operation, *parameter learning* is performed online. Short-term averaging techniques are employed to obtain run-time estimates of workload parameters based on past history. The parameter values estimated by learning are then used for addressing the lookup table and obtain the power management policy. Clearly, in many cases the estimated parameter values do not correspond exactly to values sampled in the table. If this is the case, *policy interpolation* is employed to obtain a policy as a combination of the policies in table locations corresponding to parameter values close to the estimated ones.

Experimental results reported by Chung *et al.* [41] indicate that adaptive techniques are advantageous even in the stochastic optimization framework. Simulations of power-managed systems under highly nonstationary workloads show that the adaptive technique performs nearly as well as the ideal policy computed offline, assuming perfect knowledge of workload parameters over time.

IV. Implementation of Dynamic Power Management

In this section, we address how different DPM schemes have been implemented in circuits and systems. At the same time, we describe the infrastructure that will enable the implementation of complex power management policies in electronic systems. The section is organized as follows. We describe first the physical mechanisms for power management of digital and other types of components. We review how DPM is implemented in hardware circuits that include power-manageable components. We address next system-level design, and we describe how power management is implemented in hardware/software systems, with particular reference to operating system-based power management. We conclude by presenting some experimental results on software-managed personal computers.

A. Power Management in System Components

Our working definition of system component has been provided in Section II. The complexity of a component may vary and it is irrelevant for this discussion. In Section II-A, components are considered as black boxes. Here, we are concerned with their internal structure, and we outline several techniques that can be exploited to design power-manageable components (PMC's).

1) Clock Gating: We consider first digital components that are clocked. This class of components is wide, and it includes most processors, controllers and memories. Power consumption in clocked digital components (in CMOS technology) is roughly proportional to the clock frequency and to the square of the supply voltage. Power can be saved by reducing the clock frequency (and in the limit by stopping the clock), or by reducing the supply voltage (and in the limit by powering off a component). Note that the two limiting cases (clock freezing and powering off) are applicable only to idle components. For components that are in an active state but whose response is not performance critical, power consumption can be traded off for performance by reducing the clock frequency or the supply voltage. The latter solution is usually preferred because of the quadratic dependence of power consumption on supply voltage, and it is often combined with frequency downscaling.

When considering possibly idle digital components, clock gating (or freezing) is the most common technique for power management. Namely, the clock of an idle component can be stopped during the period of idleness. Power savings are achieved in the registers (whose clock is halted) and in the combinational logic gates where signals do not propagate due to the freezing of data in registers.

Example 4.1: Clock gating has been implemented in several processors [14]–[17]. The Alpha 21 264 microprocessor uses a hierarchical clocking scheme with gated clocks [17]. In particular, the 21 264 Floating Point Unit has a controller that can freeze the clock to its components, such as the adder, multiplier, divider, etc., according to the instructions to be executed, so that the idle components do not waste power.

The PowerPC 603 processor [14] has both local and global clock control. We highlight here a feature of global clock control. When the processor is in a `Sleep` state, the clock to all units may be disabled. On the other hand, the PLL is not necessarily disabled in the `Sleep` state, so that the system controller can choose from different levels of power savings, depending on the wake-up response time requirements. For example, if a quick wake-up is required, the processor can wake up from `Sleep` in ten system clock cycles, if the PLL is active. On the other hand, for maximum power savings, the PLL can be shut off in the `Sleep` state. In this case, the wake-up time can be as long as 100 μs, to allow the PLL to relock to the external clock. □

Clock gating has a small performance overhead: the clock can be restarted by simply deasserting the clock-freezing signal. Hence, clock gating is ideally suited for implementing self-managed components. In this case, the clock is *always* stopped as soon as some custom-designed idleness detection logic signals that the component (or some of its subunits) is idle. Several CAD tools have been developed to support design with local clock (or signal) gating [8]–[12], [47]. These tools aim at generating automatically the circuit that detects idleness and that issues the signal to freeze the clock. The tools implement various methods of realizing clock gating, which differ according to the type of unit to be controlled (e.g., sequential controller, data path, pipelined circuit) and to the type of idleness being monitored (e.g., state/output pair of a sequential circuit, external observability of some signals).

Clock gating is widely used because it is conceptually simple, it has a small overhead in terms of additional circuits and often zero performance overhead because the component can transition from an idle to an active state in one (or few) cycles. The main design challenges in the implementation of clock gating are: 1) to construct an idleness-detecting circuit which is small (and thus consuming little power) and accurate (i.e., able to stop the clock whenever the component is idle) and 2) to design gated-clock distribution circuitry that introduces minimum routing overhead and keeps clock skew under tight control [13]. In some cases, as seen in the previous example, power dissipation can be further reduced by stopping not only clock distribution, but also clock generation (i.e., by stopping the master clock PLL or the internal oscillator). This choice implies non-negligible shutdown and restart delays and it is generally not automated. Sleep states where global clock generation is stopped can only be entered by issuing external commands. For processors, shutdown can be initiated by either a dedicated instruction or by asserting a dedicated signal.

2) Supply Shutdown: It is important to stress that clock-gating does not eliminate power dissipation. First, if clock gating is local, or if the clock generator is active, there is still dynamic power dissipation on the active clock circuitry. Second, leakage currents dissipate power even when all clocks are halted. As a result, the objective of achieving minimum power dissipation, as required by some battery-powered hand-held devices, may not be achieved by clock gating.

Power consumption of idle components can be avoided by powering off the unit. This radical solution requires controllable switches on the component supply line. An advantage of this approach is the wide applicability to all kind of electronic components, i.e., digital and analog units, sensors, and transducers. A major disadvantage is the wake-up time recovery time, which is typically higher than in the case of clock gating because the component's operation must be reinitialized.

When thinking of a microelectronic circuit (e.g., processor, controller), such a component is typically structured as a hierarchical compositions of subcomponents. Thus, power shutdown is applied to a selected number of subcomponents. In the case of complex circuits, usually a portion of the circuit is not powered down, so that it can run a set of minimal monitoring and control functions, and wake up the powered-down components when needed.

Example 4.2: The StrongARM SA-1100 [3] chip has two power supplies: a `VDDI` 1.5-V internal power supply and a `VDDX` 3.3-V interface voltage supply. `VDDI` powers the CPU core and the majority of the functional units on the chip (DMA controller, MMU, LCD controller, etc.). `VDDX` powers the input–output drivers, an internal 32-KHz crystal oscillator, the system control unit, and a few critical circuits.

The `Sleep` state the SA-1100 is an example of power supply shutdown. Power in `Sleep` is reduced to 0.16 mW (as opposed to 400 mW in `Run` state) by switching off the `VDDI` supply. The shutdown sequence for entering the `Sleep` state goes through three phases: 1) flush to memory all state information that should be preserved throughout the sleep period; 2) reset all internal processor state and program wakeup events; and 3) shutdown the internal clock generator. Each phase takes approximatively 30 μs. During `Sleep`, the SA-1100 only watches for preprogrammed wake-up events. Processor wake-up goes through three phases: 1) ramp-up `VDDX` and processor clock startup; 2) wait time for stabilizing processor clock; and 3) CPU boot sequence. The first two phases take, respectively, 10 and 150 ms. The third phase has negligible duration compared to the first two. The `Sleep` state can be entered either by rising a dedicated pin (called `BATT_FAULT`) or by a software procedure that writes to the power manager control register `PMCR` of the CPU. □

Power down is applicable to electrooptical and electromechanical system components, such as displays and HDD's. For systems with mechanical moving parts, like HDD's, the time constants involved in accelerating and decelerating moving parts are usually much larger than those involved in powering up and down electronic components. Furthermore, acceleration and deceleration tend to decrease the expected lifetime of the component [34]. Lifetime reduction can be seen as another cost associated with state transitions.

Example 4.3: We consider again the IBM Travelstar 14GS disk drive [4], mentioned in Example 2.2. In this component, we can highlight as main subunits: the spindle motor, the head positioning subsystem, and the host interface. The IBM Travelstar HDD has nine power states: a `spin-up` state to initialize the drive from power down, three operational states (`seek`, `write` and `read`), and five inactive states (`Performance Idle`, `Active Idle`, `Low power idle`, `Standby`, and `Sleep`). Different physical mechanisms are used to reduce power in the inactive states. In the `Performance Idle` state, all electronic components are powered while in the `Active Idle` state, some circuitry is in power saving mode, and in the `Low power idle` the head is unloaded. Whereas the spindle motor is rotating in the three idle states, the motor is spun down in the `Standby` and `Sleep` states. In the `Standby` state the host interface is active, while in the `Sleep` it is turned off.

The power consumption in the active states (in average 2.6 W) decreases in the inactive states to the values of 2, 1.3, 0.85, 0.25, and 0.1 W, respectively. Restarting the HDD requires a peak power of 5 W, due to the acceleration of the disks. Finally, note that the lower the power consumption is, the longer the corresponding wake up time is. Thus, DPM strategies need to take advantage of the low-power states while minimizing the impact on performance. □

3) Multiple and Variable Power Supplies: DPM is also applicable to components that are not idle, but whose performance (e.g., I/O delays) requirements varies with time. The implementation technology can then be based on the *slowdown* of noncritical components. The slowdown is achieved by lowering the voltage supply, such that the component becomes performance critical.

Early implementations of multivoltage chips used a static power-directed partitioning into subunits, each powered by a different supply voltage. Most often two voltage levels were used, and level shifters were employed at the border of subunits running on different supplies [44]. The extension of this approach to the realm of DPM is to enable dynamic adjustment of power supply voltage during system operation. One of the main challenges in implementing this extension is to guarantee that clock frequency tracks the speed changes caused by dynamic voltage supply adjustments.

In the pioneering work by Nielsen *et al.* [45], self-timed circuits were employed in conjunction with variable supply voltage. Self-timed circuits synchronize using local handshake signals, hence, they do not need adjustable clocks. Unfortunately, self-timed circuits are not mainstream technology. Alternative approaches employ standard synchronous logic [46], [48], [49] coupled with adjustable clocks that adapt their frequency to the speed of the critical path under different supply voltages. Another issue in systems with dynamically variable supply voltage is that they require high-efficiency dc–dc converters that can be programmed over a wide range of output voltages. Several adjustable dc–dc converters have been described in the literature [50]–[53]. The variable supply voltage approach can be complemented by dynamic threshold-voltage adjustment, achieved by controlling the body back bias [48], [49].

Dynamically varying supply voltages may be quantized [46] and thus be restricted to a finite number of values, or may take values in a continuous range. In the former case it is possible to identify a finite number of power states for the system; in the latter the concept of finite state is not applicable. State transition take a finite time because dc–dc converters cannot support arbitrarily fast changes in supply voltage.

B. System-Level Power Management Implementation

We consider DPM at the system level, and the corresponding implementation issues. Note that DPM schemes at the system level can coexist with local power management of components.

When considering electronic systems implemented in hardware, the power manager is a specialized control unit that acts in parallel and in coordination with the system control unit. In other words, the power manager may be a hardwired or microprogrammed controller, and possibly merged with the system controller. Policies based on timeouts are easily implemented by timers. Stochastic policies can be implemented by lookup tables (when stationary) or by sequential circuits. Randomized policies require the use of pseudorandom number generators, that can be implemented by *linear feedback shift registers* (LFSR's).

Typical electronic systems are software programmable, and a majority have an operating system ranging from a simple run-time scheduler or real-time operating system (RTOS) (for embedded applications) to a full-fledged operating system (as in the case of personal computers or workstations).

There are several reasons for migrating the power manager to software. Software power managers are easy to write and to reconfigure. In most cases, the designer cannot, or does not want to, interfere with and modify the underlying hardware platform. DPM implementations are still a novel art, and experimentation with software is easier than with hardware.

In general, the operating system is the software layer where the DPM policy can be implemented best. *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 codesign* 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 [20] and the *advanced configuration and power interface* (ACPI) standard proposed by Intel, Microsoft, and Toshiba [21]. 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 codesign 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.

1) Industrial Design Standards: Industrial standards have been proposed to facilitate the development of operating system-based power management. Intel, Microsoft and Toshiba proposed the open standard called *advanced configuration and power interface* (ACPI) [21]. 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* (PC's), it contains useful guidelines for a more general class of systems. The main goals of ACPI are to: 1) enable all PC's to implement motherboard dynamic configuration and power management; 2) enhance power management features and the robustness of power-managed systems; and 3) 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 Fig. 11. Applications interact with the OS kernel through *application programming interfaces* (API's). 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.

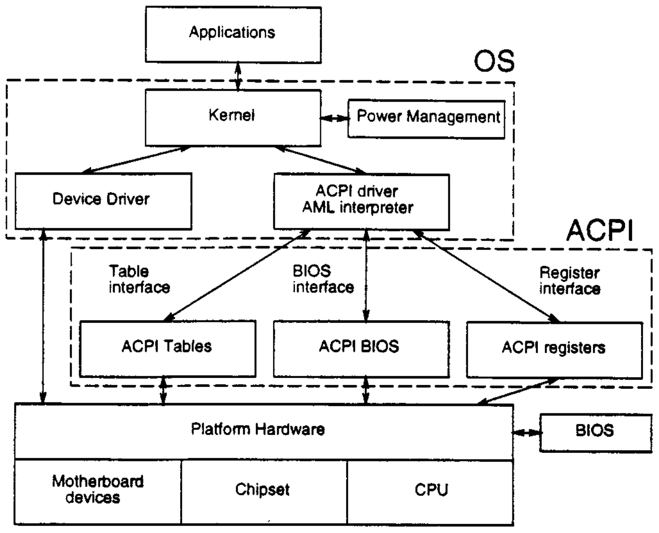


Fig. 11. ACPI interface and PC platform.

At the bottom of Fig. 11 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 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, the following.

- `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.

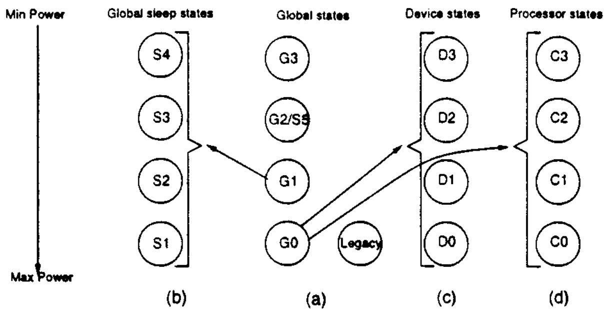


Fig. 12. State definitions for ACPI.

The global states are shown in Fig. 12(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 Fig. 12(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.

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 Fig. 12(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 Fig. 12(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.

2) ACPI-Based DPM Implementations: A set of experiments were carried out by Lu *et al.* [35], [36] to measure the effectiveness of different DPM policies. Lu used two ACPI-compliant computers, running a beta version of Windows NT V5, which is also ACPI compliant. The first computer is a VarStation 2861A desktop, using a Pentium II processor and an IBM DTTA 350–640 HDD. The second is a Sony VAIO PCG F-150 laptop, with a Pentium II and a Fujitsu MHF 2043AT HDD. The experiments aimed at controlling the HDD unit using different policies.

For this purpose, Lu implemented *filter drivers* (Fig. 13) to control the power states of the HDD's, to record disk accesses and to analyze the performance impact of the power management overhead of each algorithm. The power lines of the disks were monitored by digital multimeters, connected to a PC via a RS-232 port to record the measurements.

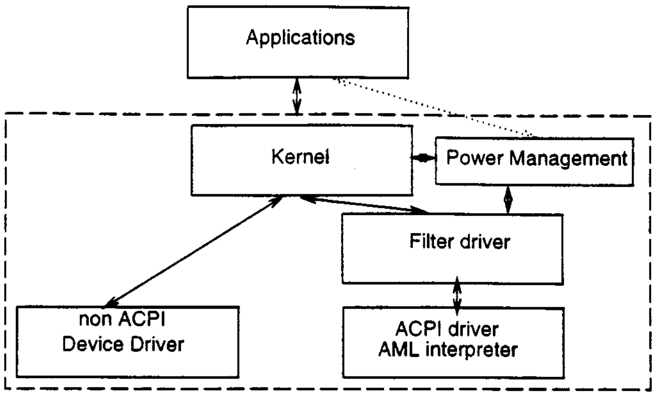


Fig. 13. DPM using filter drivers.

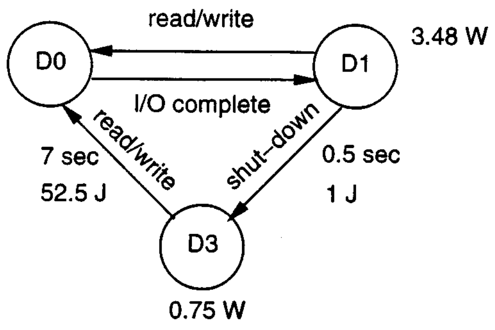


Fig. 14. PSM for IBM DTTA HDD.

TABLE I
DISK PARAMETERS: SUBSCRIPTS sd AND wu DENOTE SHUT DOWN AND WAKE UP, RESPECTIVELY

Model	P_{Off}	P_{On}	T_{sd}	E_{sd}	T_{wu}	E_{wu}
	Watt	Watt	sec	J	sec	J
IBM	0.75	3.48	0.51	1.08	6.97	52.5
Fujitsu	0.13	0.95	0.67	0.36	1.61	4.39

The IBM HDD can be in one of three states: `PowerDeviceD0` when it is reading or writing, `PowerDeviceD1` when the plates are spinning and `PowerDeviceD3` when the plates stop spinning. I/O requests only wait for seek and rotation delays when the disk is at `PowerDeviceD1` (see Fig. 14). If a request arrives when the hard disk is at `PowerDeviceD3`, it has to wait for the wake-up procedure in addition to the seek and rotation delays. The disk consumes 3.48 and 0.75 W in states `D1` and `D3`, respectively. It takes approximately 7 s and 52.5 J to wake up from `D3` to `D0`. It takes (in average) 0.5 s to enter `D3` from `D1`. The behavior of the Fujitsu HDD is similar, but with different parameters (see Table I). The break-even times of the IBM and Fujitsu HDD's are 17.6 and 5.43 s, respectively.

Experimental results are reported in [36], where a comparative analysis of different algorithms is presented. For comparison purposes, both computers execute the same trace of input data (an 11-h-long execution trace). Results show that all algorithms spend less than 1% of computation on power management itself, thus validating a fundamental premise of this body of work. For the laptop (desktop) computer, power reductions have been measured up to 55% (43%) (as compared to the always on case) and up to 34% (23%) (as compared to the default 3-min timeout policy of Windows OS). Larger power savings are achieved on the laptop computer because of the shorter break-even time of its disk.

3) Observer Implementation: As seen in Section III, power management requires information on the usage of each hardware resource, such as: 1) distribution of interarrival times of request to the resources and 2) distribution of service times for the requests. The *observer* module (Fig. 2) of the PM takes care of data collection. In ACPI-compliant PC's, the observer may rely on ACPI messages to obtain the data needed to drive the policies. However, not all computers are ACPI-compliant. In this section, we shall analyze the implementation of a power manager observer module that does not exploit ACPI, nor it is based on a proprietary Microsoft operating system. The basic requirements for the implementation of the observer are as follows.

- *Low perturbation of normal system activity*: Monitoring should be transparent to the end user and should modify the usage patterns of hardware resources as little as possible.
- *Flexibility*: It should be easy to monitor multiple types of resources. Moreover, the number and types of observed resources should be dynamically controllable. This feature is particularly useful for laptop computers where new devices can be installed during system operation (i.e., plug-and-play capability).
- *Accuracy*: Well-known system utilities give access to cumulative counts of accesses to system resources. This functionality is not sufficient to obtain accurate statistics of interarrival times and service times. One important feature of the observer is the capability of time-stamping the events with high resolution.

The software-based observer architecture analyzed in this section is called `IPM` [6], and it has been implemented as an extension of the Linux operating system [54]. The observer monitors the accesses to system resources and stores them in form of time-stamped events. The core data structure is located in kernel memory space, that is forced to reside in physical-address space. Hence, storing events in kernel space prevents the usage of memory paging, thus avoiding the severe performance penalty possibly caused by TLB misses.

On the other hand, storing the event list in kernel space imposes a tight limitation on its maximum size. The list cannot grow larger than 64 KB, which corresponds to $L_{\max} = 4096$ events. The event list is implemented as a circular buffer and it is allocated once for all (for performance reasons). The circular structure protects against memory violations. If the number of unprocessed events stored in the list grows larger than the number of slots, older events are overwritten. Event loss causes a decrease in accuracy in monitoring but does not damage normal system operation.

The size limitation of the event list in kernel memory is not a concern if events are processed and discarded as soon as they are registered (online monitoring). However, event loss should be avoided if the observer is collecting long event traces for offline processing. The observer supports offline

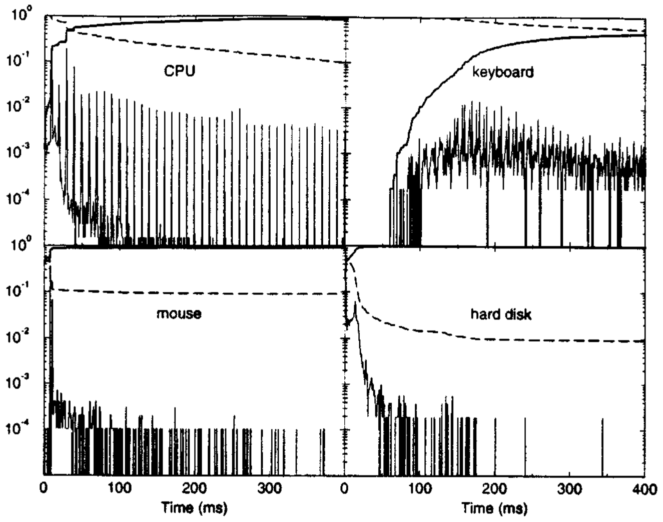


Fig. 15. Statistical analysis of the interarrival time. For each device, three curves are plotted in lin-log scale: the probability density (solid line), the probability distribution (bold line), and its complement to one (dashed line). Data refer to software development.

monitoring through a simple dumping mechanism that can be summarized as follows. Whenever the number of unprocessed events reaches a value $L_{\text{low}} < L_{\max}$, a wake-up signal is sent to a dedicated process. The process is normally inactive, waiting for the wake-up signal, thus it does not alter normal system activity. Whenever the wake-up signal is asserted, the process becomes active and can be scheduled. Clearly, the execution of this process does alter normal system activity. However, the perturbation is limited by the fact that the list is processed only when it is almost full.

Devices that are controlled by the OS through device drivers are monitored by inserting standard function calls that update the event list in the device driver routines that are run whenever the component is accessed. Monitoring does not change the flow of execution of the device driver, and it has minimal impact on the execution time. At boot time, the observer is initialized by specifying which resources should be monitored.

The CPU and all hardware components required for its operation (chipset, RAM, bus controllers, etc.) are not controlled through device drivers. Fortunately, it is possible to monitor the CPU and its ancillary components by observing that the OS kernel itself is nothing else than executable code running on the CPU. Whenever the kernel is running, the CPU is active. When there is nothing to do, the kernel schedules a dummy process, called *idle task*. Hence, to detect CPU idleness, it is sufficient to monitor the scheduling of *idle task*.

Monitor installation requires kernel recompilation, and supports monitoring of CPU, keyboard, serial and parallel ports, PS2 mouse, IDE hard disk, and CD-ROM. During the system boot, a data structure is created for each `IPM`-compliant resource, containing its name, type, configuration flags, unique identifier, and resource-specific information (such as the type of events to be monitored). Monitoring can be selectively enabled for each resource by setting the corresponding flags.

Several experiments [5] (run on a HP Omnibook 5500 CT with 133-MHz Pentium processor and 48 MB of RAM) showed that system operation is slowed down by less than 0.38% in average, even when all available system components are monitored, thus showing convincing evidence of the nonintrusiveness of the monitor. Examples of the data collected by the monitoring system are reported in Fig. 15, where the probability densities and distributions of request interarrival times are plotted for CPU, keyboard, mouse, and hard disk. Data was collected during a code development user session. Several different usage patterns were also tested (such as editing, game playing, etc.).

V. Conclusion

DPM is a powerful methodology for reducing power consumption in electronic systems. In a power-managed system, the state of operation of various components is dynamically adapted to the required performance level, in an effort to minimize the power wasted by idle or underutilized components. For most system components, state transitions have nonnegligible power and performance costs. Thus, the problem of designing power management policies that minimize power under performance constraints is a challenging one.

We surveyed several classes of power-managed systems and power management policies. Furthermore, we analyzed the tradeoffs involved in designing and implementing power-managed systems. Several practical examples of power-managed systems were analyzed and discussed in detail. Even though DPM has been successfully employed in many real-life systems, much work is required for achieving a deep understanding on how to design systems that can be optimally power managed.

Acknowledgment

The authors would like to thank E.-Y. Chung, Y.-H. Lu, G. Paleologo, and T. Šimunić at Stanford University and S. Cavallucci and A. Gordini at Bologna University for their help and useful suggestions.

References

[1] J. Lorch and A. Smith, "Software strategies for portable computer energy management," *IEEE Personal Commun.*, vol. 5, pp. 60–73, June 1998.
[2] L. Benini and G. De Micheli, *Dynamic Power Management: Design Techniques and CAD Tools*. Norwell, MA: Kluwer, 1998.
[3] *SA-1100 Microprocessor Technical Reference Manual*, Intel, 1998.
[4] *2.5-Inch Travelstar Hard Disk Drive*, IBM, 1998.
[5] L. Benini, R. Hodgson, and P. Siegel, "System-Level power estimation and optimization," in *Int. Symp. Low Power Architecture and Design*, Aug. 1998, pp. 173–178.
[6] L. Benini, A. Bogliolo, S. Cavallucci, and B. Riccó, "Monitoring system activity for OS-directed dynamic power management," in *Int. Symp. Low Power Architecture and Design*, Aug. 1998, pp. 185–190.
[7] "Advanced micro devices," in *AM29SLxxx Low-Voltage Flash Memories*, 1998.
[8] M. Alidina, J. Monteiro, S. Devadas, A. Ghosh, and M. Papaefthymiou, "Precomputation-based sequential logic optimization for low power," *IEEE Trans. VLSI Syst.*, vol. 2, pp. 426–436, Dec. 1994.
[9] S. Malik, V. Tiwari, and P. Ashar, "Guarded evaluation: Pushing power management to logic synthesis/design," in *Int. Symp. Low Power Design*, Apr. 1995, pp. 221–226.
[10] L. Benini and G. De Micheli, "Transformation and synthesis of FSM's for low power gated clock implementation," *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 630–643, June 1996.
[11] F. Theeuwen and E. Seelen, "Power reduction through clock gating by symbolic manipulation," in *Symp. Logic and Architecture Design*, Dec. 1996, pp. 184–191.
[12] M. Ohnishi *et al.*, "A method of redundant clocking detection and power reduction at RT-level design," in *Int. Symp. Low Power Electronics and Design*, Aug. 1997, pp. 131–136.
[13] J. Oh and M. Pedram, "Gated clock routing minimizing the switched capacitance," in *Design Automation and Test in Europe Conf.*, Feb. 1998, pp. 692–697.
[14] S. Gary *et al.*, "PowerPC 603, a microprocessor for portable computers," *IEEE Design & Test of Computers*, vol. 11, pp. 14–23, 1994.
[15] G. Debnath, K. Debnath, and R. Fernando, "The pentium processor-90/100, microarchitecture and low-power circuit design," in *Int. Conf. VLSI Design*, Jan. 1995, pp. 185–190.
[16] S. Furber, *ARM System Architecture*. Reading, MA: Addison-Wesley, 1997.
[17] M. Gowan, L. Biro, and D. Jackson, "Power considerations in the design of the alpha 21 264 microprocessor," in *Design Automation Conf.*, June 1998, pp. 726–731.
[18] E. Harris *et al.*, "Technology directions for portable computers," *Proc. IEEE*, vol. 83, pp. 636–657, Apr. 1996.
[19] M. Stemm and R. Katz, "Measuring and reducing energy consumption of network interfaces in hand-held devices," *IEICE Trans. Commun.*, vol. E80-B, pp. 1125–1131, Aug. 1997.
[20] Microsoft. (1997) On now: The evolution of the PC platform. [Online] http://www.microsoft.com/hwdev/pcfuture/OnNOW.HTM.
[21] Intel, Microsoft, and Toshiba. (1996) Advanced configuration and power interface specification. [Online] http://www.intel.com/ial/powermgm/specs.html.
[22] N. Bambos, "Toward power-sensitive network architectures in wireless communications: Concepts, issues and design aspects," *IEEE Personal Commun.*, vol. 5, pp. 50–59, June 1998.
[23] J. Rulnick and N. Bambos, "Mobile power management for wireless communication networks," *Wireless Networks*, vol. 3, no. 1, pp. 3–14, Jan. 1997.
[24] K. Sivalingham *et al.*, "Low-power access protocols based on scheduling for wireless and mobile ATM networks," in *Int. Conf. Universal Personal Communications*, Oct. 1997, pp. 429–433.
[25] M. Zorzi and R. Rao, "Energy-constrained error control for wireless channels," *IEEE Personal Commun.*, vol. 4, pp. 27–33, Dec. 1997.
[26] B. Mangione-Smith, "Low-power communication protocols: Paging and beyond," in *IEEE Symp. Low-Power Electronics*, Apr. 1995, pp. 8–11.
[27] P. Krishnan, P. Long, and J. Vitter, "Adaptive disk spindown via optimal rent-to-buy in probabilistic environments," in *Int. Conf. Machine Learning*, July 1995, pp. 322–330.
[28] D. Helmbold, D. Long, and E. Sherrod, "Dynamic disk spin-down technique for mobile computing," in *IEEE Conf. Mobile Computing*, Nov. 1996, pp. 130–142.
[29] F. Douglis, P. Krishnan, and B. Bershad, "Adaptive disk spin-down policies for mobile computers," in *2nd USENIX Symp. Mobile and Location-Independent Computing*, Apr. 1995, pp. 121–137.
[30] R. Golding, P. Bosh, and J. Wilkes, "Idleness is not sloth," HP Laboratories Tech. Rep. HPL-96-140, 1996.
[31] A. Karlin, M. Manasse, L. McGeoch, and S. Owicki, "Competitive randomized algorithms for nonuniform problems," *Algorithmica*, vol. 11, no. 6, pp. 542–571, June 1994.
[32] M. Srivastava, A. Chandrakasan, and R. Brodersen, "Predictive system shutdown and other architectural techniques for energy efficient programmable computation," *IEEE Trans. VLSI Syst.*, vol. 4, pp. 42–55, Mar. 1996.
[33] C.-H. Hwang and A. Wu, "A predictive system shutdown method for energy saving of event-driven computation," in *Int. Conf. Computer-Aided Design*, Nov. 1997, pp. 28–32.
[34] Y. Lu and G. De Micheli, "Adaptive hard disk power management on personal computers," in *Great Lakes Symp. VLSI*, Feb. 1999, pp. 50–53.
[35] Y. Lu, T. Šimunić, and G. De Micheli, "Software controlled power management," in *Hardware–Software Codesign Symp.*, May 1999, pp. 151–161.
[36] Y. Lu, E. Y. Chung, T. Šimunić, L. Benini, and G. De Micheli, "Quantitative comparison of power management algorithms," in *DATE, Proc. Design Automation and Test in Europe*, Mar. 2000.
[37] T. Šimunić, L. Benini, and G. De Micheli, "Event-driven power management of portable systems," in *ISSS, Proc. Int. Symp. System Synthesis*, Nov. 1999, pp. 18–23.
[38] T. Šimunić, L. Benini, P. Glynn, and G. De Micheli, "Dynamic power management of portable systems using semi-Markov decison processes," in *DATE, Proc. Design Automation and Test in Europe*, Mar. 2000.
[39] L. Benini, G. Paleologo, A. Bogliolo, and G. De Micheli, "Policy optimization for dynamic power management," *IEEE Trans. Computer-Aided Design*, vol. 18, pp. 813–33, June 1999.

[40] Q. Qiu and M. Pedram, "Dynamic power management based on continuous-time Markov decision processes," in *Design Automation Conf.*, June 1999, pp. 555–561.

[41] E. Chung, L. Benini, A. Bogliolo, and G. De Micheli, "Dynamic power management for nonstationary service requests," in *Design and Test in Europe Conf.*, Mar. 1999, pp. 77–81.

[42] S. Ross, *Introduction to Probability Models*, 6th ed. New York: Academic, 1997.

[43] M. Puterman, *Finite Markov Decision Processes*. New York: Wiley, 1994.

[44] K. Usami *et al.*, "Automated low-power technique exploiting multiple supply voltages applied to a media processor," *IEEE J. Solid-State Circuits*, vol. 33, pp. 463–472, Mar. 1998.

[45] L. Nielsen, C. Niessen, J. Sparso, and K. van Berkel, "Low-power operation using self-timed circuits and adaptive scaling of supply voltage," *IEEE Trans. VLSI Syst.*, vol. 2, pp. 425–435, Dec. 1994.

[46] A. Chandrakasan, V. Gutnik, and T. Xanthopoulos, "Data driven signal processing: An approach for energy efficient computing," in *Int. Symp. Low Power Electronics and Design*, Aug. 1996, pp. 347–352.

[47] H. Kapadia, G. De Micheli, and L. Benini, "Reducing switching activity on datapath buses with control-signal gating," in *Custom Integrated Circuit Conf.*, May 1998, pp. 589–592.

[48] K. Suzuki *et al.*, "A 300 MIPS/W RISC core processor with variable supply-voltage scheme in variable threshold-voltage CMOS," in *Custom Integrated Circuits Conf.*, May 1997, pp. 587–590.

[49] K. Usami *et al.*, "Design methodology of ultra low-power MPEG4 codec core exploiting voltage scaling techniques," in *Design Automation Conf.*, June 1998, pp. 483–488.

[50] A. Stratakos, S. Sanders, and R. Brodersen, "A low-voltage CMOS dc–dc converter for a portable battery-operated system," in *Power Electronics Specialists Conf.*, June 1994, pp. 619–626.

[51] G. Wei and M. Horowitz, "A low power switching power supply for self-clocked systems," in *Int. Symp. Low Power Electronics and Design*, Aug. 1996, pp. 313–317.

[52] W. Namgoong, M. Yu, and T. Meng, "A high-efficiency variable-voltage CMOS dynamic dc–dc switching regulator," in *Int. Solid-State Circuits Conf.*, Feb. 1997, pp. 380–381.

[53] V. Gutnik and A. Chandrakasan, "Embedded power supply for low-power DSP," *IEEE Trans. VLSI Syst.*, vol. 5, pp. 425–435, Dec. 1997.

[54] L. Torvalds, "The Linux operating system," *Commun. ACM*, vol. 42, no. 4, pp. 38–39, Apr. 1999.

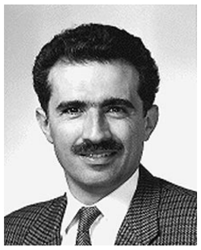
Luca Benini (M'93) received the Dr.Eng. degree in electrical engineering from the University of Bologna, Bologna, Italy, in 1991 and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, in 1994 and 1997, respectively.

Since 1998, he has been an Assistant Professor in the Department of Electronics and Computer Science, University of Bologna. He also holds visiting professor positions at Stanford University and Hewlett-Packard Laboratories, Palo Alto, CA. His research interests are in all aspects of computer-aided design of digital circuits, with special emphasis on low-power applications and in the design of portable systems.

Dr. Benini has been a member of technical program committees for several technical conferences, including the Design and Test in Europe Conference and the International Symposium on Low Power Design.

Alessandro Bogliolo (M'95) received the Laura degree in electrical engineering and the Ph.D. degree in electrical engineering and computer science from the University of Bologna, Bologna, Italy, in 1992 and 1998, respectively.

From 1992 to 1999, he was with the Department of Electronics, Computer Science and Systems (DEIS), University of Bologna. In 1995 and 1996, he was a Visiting Scholar at the Computer Systems Laboratory (CSL), Stanford University, Stanford, CA. Since then he has cooperated with the research group of Prof. De Micheli at Stanford. In 1999, he joined the Department of Engineering (DIF), University of Ferrara, Ferrara, Italy, as an Assistant Professor. His research interests are in the area of computer-aided design of digital integrated circuits and systems, with particular emphasis on high-level power modeling, power optimization, and intellectual property protection.



Giovanni De Micheli (F'94) is a Professor of Electrical Engineering and Computer Science at Stanford University, Stanford, CA. His research interests include several aspects of the computer-aided design of integrated circuits and systems, with particular emphasis on automated synthesis, optimization, and validation. He is the author of *Synthesis and Optimization of Digital Circuits* (New York: McGraw-Hill, 1994) and a coauthor of *Dynamic Power Management: Circuit Techniques and CAD Tools* (Norwell, MA: Kluwer, 1998) and three other books. He is the Editor-in-Chief of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN.

Dr. De Micheli received the 1987 IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN/ICAS Best Paper Award, a Presidential Young Investigator Award in 1988, and two Best Paper Awards at the Design Automation Conference in 1983 and in 1993. He is Vice President (for publications) of the IEEE CAS Society. He is the General Chair of the 37th Design Automation Conference. He was Program and General Chair of the International Conference on Computer Design (ICCD) in 1988 and 1989, respectively. He was also Codirector of the NATO Advanced Study Institutes on Hardware/Software Co-design, Tremezzo, Italy, in 1995 and the Logic Synthesis and Silicon Compilation, L'Aquila, Italy, in 1986.