System-Level Power Management: An Overview

Ali Iranli and Massoud Pedram
University of Southern California
Dept of Electrical Engineering
Los Angeles CA

Abstract

One of the key challenges of computer system design is the management and conservation of energy. This challenge is evident in a number of ways. The goal may be to extend the battery lifetime of a portable, battery-powered device. The processing power, memory, and network bandwidth of such devices are increasing quickly, resulting in an increase in demand for higher power dissipation, while the battery capacity is improving at a much slower pace. Other goals may be to limit the cooling requirements of a computer system or to reduce the financial cost of operating a large computing facility with a high energy bill. This chapter focuses on techniques which dynamically manage electronic systems in order to minimize its energy consumption. Ideally, the problem of managing the energy consumed by electronic systems should be addressed at all levels of design, ranging from low-power circuits and architectures to application and system software capable of adapting to the available energy source. Many research and industrial efforts are currently underway to develop low-power hardware as well as energy-aware application software in the design of energy-efficient computing systems. Our objective in this chapter is to explore what the system software, vis-à-vis the operating system (OS), can do within its own resource management functions to improve the energy efficiency of the computing system without requiring any specialized, low-power hardware or any explicit assistance from application software and compilers. There are two approaches to consider at the OS-level for attacking most of the specific energy-related goals described above. The first is to develop resource management policies that eliminate waste or overhead and allow energy-efficient use of the devices. The second is to change the system workload so as to reduce the amount of work to be done, often by changing the fidelity of objects accessed, in a manner which will be acceptable to the user of the application. This chapter provides a first introduction to these two approaches with appropriate review of related works.

Background

A system is a collection of components whose combined operation provides a useful service. Typical systems consist of hardware components integrated on single or multiple chips and various software layers. Hardware components are macro-cells that provide information processing, storage, and interfacing. Software components are programs that realize system and application functions. Sometimes, system specifications are required to fit into specific interconnections of selected hardware components (e.g., Pentium processor) with specific system software (e.g., Windows or Linux) called *computational platforms*.

System design consists of realizing a desired functionality while satisfying some design constraints. Broadly speaking, constraints limit the design space and relate the major design trade-off between *quality of service* (QoS) versus cost. QoS is closely related to performance, i.e., system throughput and/or task latency. QoS relates also to the system *dependability*, i.e., to a class of system metrics such as reliability, availability, and safety that measure the ability of the system to deliver a service correctly, within a given time window and at any time. Design cost relates to design and manufacturing costs (e.g., silicon area, testability) as well as to operation costs (e.g., power consumption, energy consumption per task.)

In recent years, the design trade-off of performance versus power consumption has received large attention because of: (i) the large number of systems that need to provide services with the energy provided by a battery of limited weight and size, (ii) the limitation on high-performance computation because of heat dissipation issues, and (iii) concerns about dependability of systems operating at high temperatures because of power dissipation. Here we focus on *energy-managed computer* (EMC) Systems. These systems are characterized by one or more high-performance processing cores, large on-chip memory cores, various I/O controller cores. The use of these cores will force system designers to treat them as black boxes and abandon the detailed tuning of their performance/energy parameters. On the other hand, various I/O devices are provisioned in the system level design to maximize the interaction between the user and the system and/or among different users of the same system.

Dynamic power management (DPM) is a feature of the run-time environment of an EMC system that dynamically reconfigures itself to provide the requested services and performance levels with a minimum number of active components or a minimum activity level on such components. 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.) The fundamental premise for the applicability of DPM is that systems (and their components) experience non-uniform workloads during operation time. Such an assumption is valid for most systems, both when considered in isolation and when inter-networked. A second assumption of DPM is that it is possible to predict, with a certain degree of confidence, the fluctuations of workload. In this chapter we present and classify different modeling frameworks and approaches to dynamic power management.

Modeling Energy Managed Computers (EMC)

An EMC models the electronic system as a set of interacting power manageable components (PMC's) controlled by one or more *power managers* (PM's.) We model PMC's as black boxes. We are not concerned on how PMC's are designed; instead we focus on how they interact with each other and the operating 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 system wide energy management policies. We consider PMC's in isolation first. Next, we describe DPM for systems with several interacting components.

A. Power Manageable Components

A *PMC* is defined to be an atomic block in an electronic system. PMCs can be as complex as a printed circuit board realizing an I/O device, or as simple as a functional unit within a chip. At the system level, a component is typically seen as a black box, i.e., no data is available about its internal architecture. The key attribute of a PMC is the availability of multiple modes of operation, which span the power-performance tradeoff curve. Non-power-manageable components are designed for a given performance target and power dissipation specification. 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 so as to provide just enough computational capability to meet a target timing constraint while minimizing the total energy consumption of completing a computational task. In the limit, one can think of a PMC to have a continuous range of operational modes. Clearly, as the number of available operational modes increases the ability to perform fine-grained control of the PMC to minimize the power waste and achieve a certain performance level increases. In practice, the number of modes of operation tends to be small because of the increased design complexity and hardware overhead of supporting multiple power modes.

Another important factor about a PMC is the overhead associated with the PMC transitioning from one mode of operation to next. Typically, this overhead is expressed as transition energy and a delay penalty. If the PMC is not operational during the transition, some performance is lost whenever a transition is initiated. The transition overhead depends on PMC implementation: in some cases the cost may be negligible, but, generally, it is not. Transition overhead plays a significant role in determining the number and type of operational modes enabled by the PMC designer. For example, excessive energy and delay overheads for transitions into and out of a given PMC state may make that state nearly useless because it will be very difficult to recompense the overheads unless the expected duration of contiguous time that the PMC remains in that state is especially long.

Mathematically, one can represent a PMC by a finite state machine where states denote the operational modes of the PMC and state transitions represent mode transition. Each edge in the state machine has an associated energy and delay cost. In general, low-power states have lower performance and larger transition overhead compared to high-power states. This abstract model is referred to as a *Power State Machine* (PSM.) Many single-chip components like processors [1] memories [2], and archetypal I/O devices such as disk drives [3], wireless network interfaces [4], and displays [5] can readily be modeled by a PSM.

Example: The StrongARM SA-1100 processor [6] is an example of a PMC. It has three modes of operation: *RUN*, *STDBY*, and *SLEEP*. The *RUN* mode is the normal operating mode of the SA-1100: every on-chip resource is functional. The chip enters the *RUN* mode after successful power-up and reset. *STDBY* mode allows a software application to stop the CPU when it is not in use, while continuing to monitor interrupt requests on or off chip. In the *STDBY* mode, the CPU can be brought back to the *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 *STDBY*, the SA-1100 performs an orderly shutdown of its 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 Figure 1. States are marked with power dissipation and performance values, edges are marked with transition times and energy dissipation overheads. The power consumed during transitions is approximately equal to that in the *RUN* mode. Notice that both *STDBY* and *SLEEP* have null performance, but the time for exiting *SLEEP* is much longer than that for exiting *STDBY* (10s versus 160ms.) On the other hand, the wake-up time from the *SLEEP* state is much larger, and therefore, it must 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 which would shut down the processor as soon as an idle period was detected tends to reduce performance without actually saving any power (the ratio of the energy consumption divided by the transition time associated with any of the state transitions is of the same order of power dissipation in the RUN state.) An external PM that controls the inter-mode transitions of the SA-1100 processor must observe the workload and make 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 can keep the processor nearly always in the *SLEEP* state. ♦

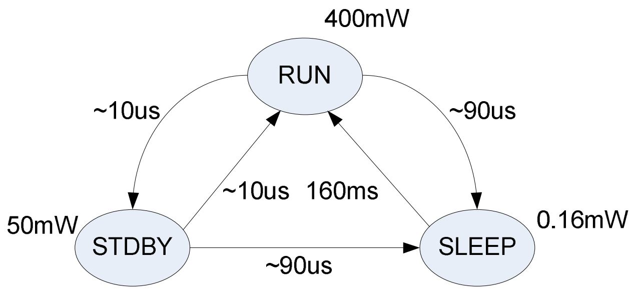


Figure 1. Power State Machine for Strong ARM SA1100

B. Dynamic Power Management Techniques

This section reviews various techniques for controlling the power state of a system and its components. One may consider components as black boxes, whose behavior is abstracted by the PSM model and focus on how to design effective power management policies. Without loss of generality, consider the problem of controlling a single component (or, equivalently, the system as a whole.) Furthermore, assume that transitions between different states are instantaneous and the transition energy overhead is non-existent. In such a system, DPM is a trivial task and the optimum policy is greedy one i.e., 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 non-negligible performance and power costs for 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 in order to 1) turn on and stabilize the power supply and the clock; 2) reinitialize the system; and 3) restore the context. When power state transitions have a cost, finding the optimal DPM policy becomes a difficult optimization problem. In this case, the DPM policy optimization is equivalent to a decision making problem in which the PM must decide if and when it is worthwhile (from a performance and power dissipation viewpoint) to transition to which low-power state (in case of having multiple low-power states.)

Example: Consider the StrongARM SA-1100 processor described in previous example. Transition times between *RUN* and *STDBY* states are very fast so that the *STDBY* 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 longer and has to be compared with the time constants for the workload variations to determine whether or not the processor should be shut down. In the limiting case of a workload without any idle period longer than the time required to enter and exit the *SLEEP* state, a greedy policy for shutting down the processor (i.e., moving to *SLEEP* state as soon as an *STDBY* period is detected) will result in performance loss, but no power saving. This is because the power consumption associated with state transitions is of the same order of magnitude as that of the *RUN* state. An external PM which controls the power state transitions of the SA-1100 processor must make online decisions based on the workload and target performance constraints. ♦

The aforementioned example was a simple DPM optimization problem, which illustrated the two key steps of designing a DPM solution. The first task is the *policy optimization,* which is the problem of solving a power optimization problem under performance constraints. The second task is the *workload prediction*, which is the problem of predicting the near-future workload. In the following, different approaches for implementing these two problems will be discussed.

The early works on DPM focused on predictive shutdown approaches [7][8] which make use of “time-out” based policies. A power management approach based on discrete time Markovian decision processes was proposed in [9]. The discrete-time model requires policy evaluation at periodic time instances and may thereby consume a large amount of power even when no change in the system state has occurred. To surmount this shortcoming, a model based on continuous-time Markovian decision processes (CTMDP) was proposed in [10]. The policy change under this model is asynchronous and is thus more suitable for implementation as part of a real-time operating system environment. Reference [11] proposed time-indexed semi-Markovian decision processes for system modeling. Other approaches such as adaptive learning based strategies [12], session clustering and prediction strategies [13], on-line strategies [14][15], and hierarchical system decomposition and modeling [34] have also been utilized to find a DPM policy of EMCs.

In the following sections, we describe various DPM techniques in more detail.

B.1 Predictive Shutdown Approaches

Applications such as display severs, user interface functions, and communication interfaces are “*event-driven*’’ in nature with intermittent computational activity triggered by external events and separated by periods of inactivity. An obvious way to reduce average power consumption in such applications is to shut the system down during periods of inactivity. This can be accomplished by shutting off the system clock or in certain cases by shutting off the power supply (cf. Figure 2.)

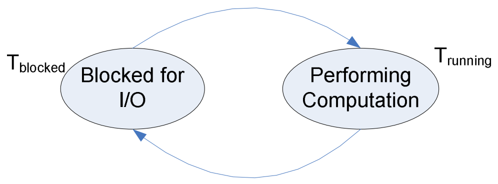


Figure 2. Event-driven application alternates between blocked and running states

An event-driven application will alternate between a blocked state where it stalls the CPU waiting for external events and a running state where it executes instructions. Let $T_{blocked}$ and $T_{running}$ denote the average time spent in the blocked and the running states, respectively. One can improve the energy efficiency by as much as a factor of $1+\frac{T_{blocked}}{T_{running}}$ provided that the system is shut down whenever it is in the blocked state.

There are two key questions: 1) how to shutdown, and 2) when to shutdown. The first question is addressed by developing mechanisms for stopping and restarting the clock or for turning off and on the power supply. The second question is addressed by devising policies such as “shut the system down if the user has been idle for five minutes.” Although these two issues are not really independent because the decision about when to shutdown depends on the overhead of shutting down the system, the predictive shutdown approaches focus primarily on the question of deciding when to shutdown while being aware of the available shutdown mechanisms. Simple shutdown techniques, for example shutting down after a few seconds of no keyboard or mouse activity, are typically used to reduce power consumption in current notebook computers. However, the event-driven nature of modern applications, together with efficient hardware shutdown mechanisms provided by PMCs, suggests the possibility of a more aggressive shutdown strategy where parts of the system may be shutdown for much smaller intervals of time while waiting for events.

In this section we explore a shutdown mechanism where we try to predict the length of idle time based on the computation history, and then shut the processor down if the predicted length of the idle time justifies the cost in terms of both energy and performance overheads of shutting down. The key idea behind the predictive approach can be summarized as follows: "Use history to predict whether $T_{blocked}$ will be long enough to justify a shutdown." Unfortunately, this is a difficult and error-prone task. One therefore has to resort to heuristics to predict $T_{blocked}$ for the near future. References [7][8] present approaches where based on the recent history, a prediction is made as to whether or not the next idle time will be long enough to at least break even with the shutdown overhead. Results demonstrate that for reasonable values of the shutdown overhead, the predictive approaches tend to result in sizeable energy savings compared to the greedy shutdown approach, while the performance degradation remains negligible.

Restricting the analysis to a simple event-driven model of an application program running on the SA-1100 processor and considering internally controlled transfer between RUN and STDBY states and externally controlled transfer between STDBY and SLEEP states of the processor, the system can be modeled by a partially self-power-managed PSM with only two states (cf. Figure 3): ON and OFF. The ON state is a macro-state representing the RUN and STDBY states of the processor and the local policy used by the processor itself to move between RUN and STNDBY states depending on the workload. The OFF macro-state is the same as the SLEEP state. The power consumption associated with the ON state is the expected power consumption in this macro-state and is calculated as a function of standby time, local transition probabilities and energy overhead of the transitions. Transitions between ON and OFF macro-states correspond to transitions between RUN and SLEEP states and their overheads are set accordingly.

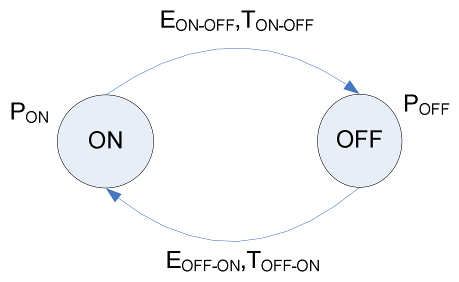


Figure 3. PSM of a two-state power-manageable component.

The processor starts in the ON state, and makes transitions from ON to OFF back to ON state. Let $T_{ON}[i]$ and $T_{OFF}[i]$ denote the time spent by the application in the i^{th} visit to the ON and the OFF states, respectively. Furthermore, we define T_{ON} as the average of $T_{ON}[i]$ over all i, and similarly T_{OFF} as the average of $T_{OFF}[i]$ over all i. Let T_{ON-OFF} and E_{ON-OFF} denote the time and energy dissipation overhead associated with the transfer to OFF state. T_{OFF-ON} and E_{OFF-ON} are similarly defined (cf. Figure 4.) In predictive shut down approaches the PM predicts the upcoming duration of time for which the system will be idle, $T_{OFF}[i]$, based on the information from the current active period, $T_{ON}[i]$ and previous active and idle periods $T_{ON}[j]$ and $T_{OFF}[j]$ for $j=i-1$, $i-2$, … 1. The policy then is to transfer the processor from ON to OFF state if $T_{OFF}[i] \geq T_{BE}$, where T_{BE} is the duration of break-even time. The processor is then turned on (it moves from OFF to ON state) as soon as a new request for data processing comes in.

In [7], the authors proposed two different approaches for predicting $T_{OFF}[i]$. The first approach uses regression analysis on application traces and calculates $T_{OFF}[i]$ in terms of $T_{OFF}[i-1]$ and $T_{ON}[i]$. Notice that $T_{OFF}[i-1]$ denotes the actual (and not the initially predicted) duration of the OFF time on the $(i-1)^{st}$ visit to the OFF state. For their second approach, the authors simplify the analysis based on the observation that long OFF periods are often followed by short ON periods. Therefore, a simple rule is constructed whereby, based on the duration of the current ON period, $T_{ON}[i]$, the PM predicts the duration of the next OFF period, $T_{OFF}[i]$, to be larger or smaller than the break-even time, T_{BE}, and therefore, decides whether it should maintain or change the current power state of the processor.

Reference [8] improves this approach by using an *exponential-average* of the previous OFF periods as the predicted value for the upcoming idle period duration. More precisely,

$$\begin{aligned} T_{Off}^{est}[i] &= a \cdot T_{Off}^{act}[i-1] + a \cdot (1-a) \cdot T_{Off}^{act}[i-2] + a \cdot (1-a)^2 \cdot T_{Off}^{act}[i-3] + \cdots + \\ & a \cdot (1-a)^{m-2} \cdot T_{Off}^{act}[i-m+1] + (1-a)^m \cdot T_{Off}^{est}[i-m] \end{aligned} \qquad (1)$$

where $0 \le a \le 1$ is a weighting coefficient. Parameter a controls the relative weight of recent and past history in the prediction. If $a = 0$, then $T_{OFF}^{est}[i] = T_{OFF}^{act}[i-m]$, i.e., the recent history has no effect on the estimation. On the other hand, if $a = 1$, then $T_{OFF}^{est}[i] = T_{OFF}^{act}[i-1]$, i.e., only the immediate past matters in setting the duration of the next idle period. In general, however, this equation favors near past historical data. For example, for $a = ½$, $T_{OFF}^{act}[i-1]$ has a weight of ½ whereas $T_{OFF}^{act}[i-3]$ has a weight of ⅛ .

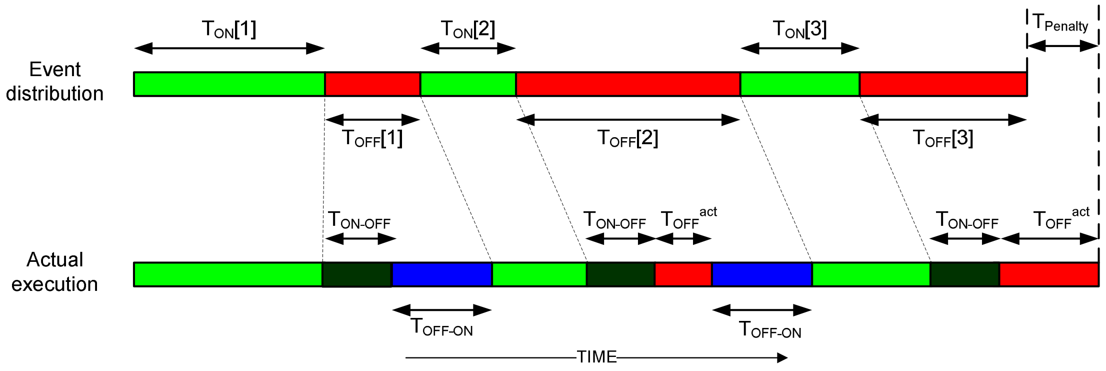


Figure 4. Graphical illustration of how a simple greedy algorithm can result in a significant delay penalty.

As mentioned before, when the PMC resumes the ON state from the OFF state (i.e., on system wake-up), the PMC suffers a delay penalty of $T_{OFF\text{-}ON}$ having to restore the original system state. This delay penalty can have a large negative impact on the PMC's performance. Reference [8] circumvents this problem by proposing a pre-wakeup approach before the arrival time of the next event. In this approach, the system starts the activation process immediately after the predicted time interval for the current idle period. Let's consider the case where the predicted idle period is overestimated i.e., $T_{Off}^{est}[i] = T_{Off}^{act}[i] + d$ for $d > 0$. Two sub-cases are possible 1.1) $d \le T_{OFF\text{-}ON}$: the system wakes up after $T_{OFF\text{-}ON}$ and the delay penalty is ($T_{OFF\text{-}ON} - d$); 1.2) $d > T_{OFF\text{-}ON}$: the system is awakened after $T_{OFF\text{-}ON}$ time units. Next, consider the case where the predicted idle period is underestimated i.e., $T_{Off}^{est}[i] = T_{Off}^{act}[i] - d$ for $d \ge 0$. Again we consider two-sub-cases: 2.1) $d \le T_{OFF\text{-}ON}$: the system will wake up after $T_{OFF\text{-}ON}$ and immediately starts executing the arrived computational task. There is no energy waste and the delay penalty is ($T_{OFF\text{-}ON} - d$). 2.2) $d > T_{OFF\text{-}ON}$: the system will wake up after $T_{OFF\text{-}ON}$ and remain ON for a period of ($d - T_{OFF\text{-}ON}$) time units ahead of the next required computation. Energy waste is $(d - T_{OFF\text{-}ON}).P_{ON}$. There is no delay penalty. In summary, the pre-wakeup policy results in shorter delay penalty in sub-cases 1.1, 2.1, and 2.2, but it results in energy waste in subcase 2.2.

To alleviate the chances for under-estimation of the idle period, reference [8] proposes a timeout scheme which periodically examines the PMC to determine whether it is idle but not shut down. If that is the case, then it increases $T_{OFF}^{est}[i]$. The chance of over-prediction is reduced by imposing a saturation condition on predictions, i.e., $T_{OFF}^{est}[i] \le C_{\max} \cdot T_{OFF}^{est}[i-1]$.

Several other adaptive predictive techniques have been proposed to deal with non-stationary workloads. In the work by Krishnan et al. [16], a set of prediction values for the length of ile period is maintained. In addition, each prediction value is annotated with an indicator to show how successful it would have been if it had been used in the past. The policy then chooses for the length of next idle period the prediction which has the highest indicator value among the set of available ones. Another policy, presented by Helmbold et al. [17], also keeps a list of candidate predictions and assigns a weight to each timeout value based on how well it would have performed for past requests relative to an optimum offline strategy. The actual prediction is then obtained as a weighted average of all candidate predictions. Another approach, introduced by Douglis et al. [18], keeps only one prediction value but adaptively changes the value. In particular, it increases (decreases) the prediction value when this value causes too many (few) shutdowns.

The accuracy of workload prediction can be increased by customizing predictors to a particular class of workloads. This kind of customization restricts its scope of applicability, but also reduces the difficulties of predicting completely general workloads. A recently proposed adaptive technique [19], which is specifically tailored toward hard-disk power management, 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 only used to predict the session length. Prediction of a single parameter is easily accomplished and the reported accuracy is high.

As mentioned earlier, there are periods of unknown duration, during which there are no tasks to run and the device can be powered down. These idle periods end with the arrival of a service request. The decision that the online DPM algorithm has to make is when to transition to a lower power state, and which state to transition to. The power-down states are denoted by $s_0,\ldots, s_k$, with associated decreasing power consumptions of $P_0,\ldots, P_k$. At the end of the idle period, the device must return to the highest power state, s_0. There is an associated transition energy e_{ij} and transition time t_{ij} to move from state s_i to s_j. The goal is to minimize the energy dissipation consumed during the idle periods. Online power-down techniques can be evaluated according to a competitive ratio (a ratio of 1 corresponds to the optimal solution.) There is a deterministic 2-competitive algorithm for two-state systems, which keeps the service provider in the active state until the total energy consumed is equal to the transition energy. It is recognized that this algorithm is optimally competitive. Furthermore, if the idle period is generated by a known probability distribution, then there is an optimally competitive probability-based algorithm which is ($e/(e-1)$)-competitive [20]. For some systems, the energy needed and time spent to go from a higher power state to a lower power state is negligible. Irani et al. show in [14] that for such systems, the two-state deterministic and probability-based algorithms can be generalized to systems with multiple sleep states so that the same competitive ratios can be achieved. The probability-based algorithm requires information about a probability distribution, which generates the length of the idle period. In [15], Irani et al. give an efficient heuristic for learning the probability distribution based on recent history.

B.2 Markovian Decision Process-Based Approaches

The most aggressive predictive power management policies turn off every PMC as soon as it becomes idle. Whenever a component is needed to carry out some task, the component must first be turned on and restored to its fully functional state. As mentioned above, the transition between the inactive and the functional state has latency and energy overheads. As a result, "eager" policies are often unacceptable because they can degrade performance without decreasing power dissipation. The heuristic power management policies are useful in practice although no strong optimality result has been proved for these types of policies. On the other hand, stochastic control based on Markov models has emerged as an effective power management framework. In particular, the stochastic PM techniques have a number of key advantages over predictive techniques. First, they capture a global view of the system, thus allowing the designer to search for a global optimum which can exploit multiple inactive states of multiple interacting resources. Second, they compute the exact solution (in polynomial time) for the performance-constrained power optimization problem. Third, they exploit the vigor and robustness of randomized policies. However, a number of key points must be considered when deciding whether or not to utilize a stochastic DPM technique. First, the performance and power obtained by a policy are expected values, and there is no guarantee that the results will be optimum for a specific workload instance (i.e., a single realization of the corresponding stochastic process.) Second, policy optimization requires a priori Markov models of the service provider (SP) and service requester (SR.) One can safely assume that the SP model can be pre-characterized; however, this assumption may not be true about the SR's model. Third, policy implementation tends to be more involved. An implicit assumption of most DPM techniques is that the power consumption of the PM is negligible. This assumption must be validated on a case-by-case basis, especially for stochastic approaches. Finally, the Markov model for the SR or SP may be only an approximation of a much more complex stochastic process. If the model is not accurate, then the "optimal" policies are also approximate solutions.

In the following, we consider a discrete-time (i.e., slotted time) setting [9]. Time is described by an infinite series of discrete values $t_n = T.n$, where T is the time resolution (or period), and $n \in \mathbb{N}^+$. The EMC is modeled with a single SR (or user) whose requests are en-queued in a single queue, service queue (SQ), and serviced by a single SP. The PM controls over time the behavior of the SP (Figure 5.)

Service Requester (SR): This unit sends requests to the SP. The SR is modeled as a Markov Chain whereby the observed variable is the number of requests s_r sent to the SP during time period t_n. The service request process and its relevant parameters are known. Moreover, it is known that in each time period a maximum of S_p requests can be generated.

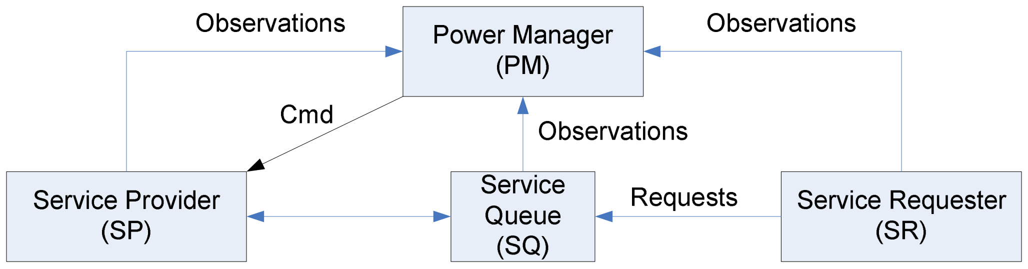


Figure 5. Illustration of the abstract system model

Example: Consider a "bursty" workload with a maximum of one request per period, i.e., the SR has two states as depicted in Figure 6. Since the workload comes in bursts, a request will be generated at time t_{n+1} with a probability of 0.85 if a request is received at time t_n. On the other hand, if there is no request at time t_n, then with a probability of 0.92 there will be no request at time t_{n+1}. The mean duration of a stream of requests is equal to 1/0.15 =6.67 periods. ♦

,

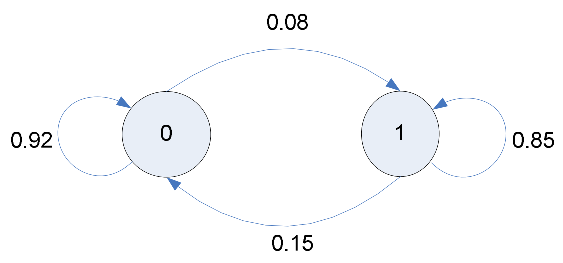
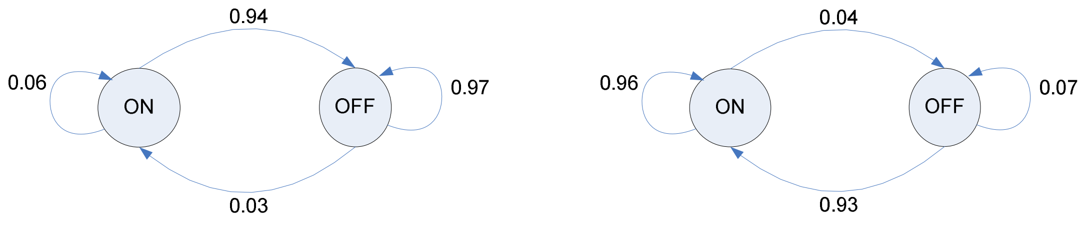


Figure 6. A Bursty Service Requester's Stochastic Model

Service Provider (SP): The SP is a PMC which services requests issued by the SR. In each time period, the SP can be in only one state. Each state $s_p \in \{1, \cdots, S_P\}$ is characterized by a performance level and by a power consumption level. In the simplest example, one could have two states: *ON* and *OFF*. In each period, transitions between power states are controlled by a power manager (PM) through commands: $cmd \in CMD = \{1, \cdots, N_C\}$. For example, one can define two simple commands: *Go2Active* and *Go2Sleep*. When a specific command is issued, the SP will move to a new state with a fixed probability depending on the command *cmd*, and on the departing and arriving states. In other words, when a command is issued by the PM, there is no guarantee that the command is immediately executed. Instead, the command influences the way in which the SP will act in the future. This probabilistic model describes the view where the evolution in time of power states is modeled by a Markov process in which the transition probability matrix is dependent on the commands issued by the PM. In other words, there is one transition probability matrix for each command *cmd*.

Back to the SA-1100 example with two states, Figure 7 depicts the probabilistic behavior of the device under influence of *Go2Sleep* and *Go2Active* commands:



a. stochastic model of the PMC when *Go2Sleep* command has been issued

b. stochastic model of the PMC when *Go2Active* command has been issued

Figure 7. Stochastic model of the SA-1100

The transition time from *OFF* to *ON* when *Go2Active* has been issued is a geometric random variable with an average of 1/0.04 = 25 time periods. Each power state has a specific power consumption rate and performance (e.g., clock speed), which is a function of the state itself. In addition, each transition between two power states in annotated with an energy cost and a latency, representing the overhead of transitioning between the two corresponding states. Such information is usually provided in the data-sheets of the PMCs.

Service Queue (SQ)**:** When service requests arrive during one period, they are buffered in a queue of length ($S_q \geq 1$.) The queue is usually considered to be a FIFO, although other schemes can also be modeled efficiently. The request is processed and serviced within the period with a probability dependent on the power state of the SP. In this way, the model captures the non-deterministic service time of a request as a geometric random variable, similar to the exponential service time for the G/M/1 class in the queuing theory [21]. It follows that also the queue length (denoted by s_q, with $0 \le s_q < S_q$) is a Markov process with transition matrix $P_{SQ}(s_p, s_r)$. Again back to the example, if a request is serviced with probability 0.9 in the *ON* state and with probability zero in the *OFF* state and the buffer can contain at most one request, then the transition matrix $P_{SQ}(s_p, s_r)$ will be

$$P_{SP}(ON,0) = \begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 1.0 & 0.0 \\ 1 & 0.9 & 0.1 \end{array} \qquad P_{SP}(ON,1) = \begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 0.9 & 0.1 \\ 1 & 0.9 & 0.1 \end{array}$$

$$P_{SP}(OFF,0) = \begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 1.0 & 0.0 \\ 1 & 0.0 & 1.0 \end{array} \qquad P_{SP}(OFF,1) = \begin{array}{c|cc} & 0 & 1 \\ \hline 0 & 0.0 & 1.0 \\ 1 & 0.0 & 1.0 \end{array} \qquad (2)$$

Power Manager (PM): This component communicates with the SP and attempts to set its state at the beginning of each period by issuing commands from among a finite set, *CMD*. This goal is in turn achieved only in a probabilistic sense, that is, the PM changes the transition matrix of the SP by issuing a particular command. In the aforementioned example, the two possible commands are *Go2Active*, and *Go2Sleep*. The PM has all specifications and collects all relevant information (by observing SR, SQ, and SP) needed for implementing a power management policy. The power consumption of the PM is assumed to be much smaller than that of the PMCs it controls and so it is not a concern. The state of the EMC composed of the SP, the SR and the queue is then a triplet, $s = (s_p, s_q, s_r)$. Being the composition of three Markov chains, s is a Markov chain (with $S = S_r \times S_p \times S_q$ states), whose transition matrix P(*cmd*) depends on the command *cmd* issued to the SP by the PM. Hence, the system is fully described by a set of N_C transition matrices, one for each command.

In the above description no mention is made of the energy source (i.e., the battery.) In the stochastic approaches the goal is to minimize (or bound) the average power consumption of the SP, and not to maximize the expected battery life. This choice has several advantages: it does not need to consider the details of the power source (rate-dependent energy discharge characteristics, and energy recovery), while still retains the primary feature of minimizing (or constraining) the power consumption level. However, there have been recent attempts to incorporate the battery behavior in modeling the EMC systems while finding a solution to DPM problem [22].

At the beginning of each time period t_n, the PM observes the “history” of the system, i.e., the sequence of states and commands up to t_{n-1}. It then controls the SP by making a decision. In *deterministic policies*, the PM makes a single decision to issue specific command on the basis of the history of the system. On the other hand, in the much broader set of policies called the *randomized policies*, PM assigns probabilities to every available command and then chooses the command to issue, according to this probability distribution. In this way, even if the same decision is taken in different periods, the actual commands issued by the PM can be different.

Mathematically speaking, let H_n represent history of the EMC, then a decision $\delta(H_n)$ in a randomized policy is a set of probabilities, p_{cmd}, where each p_{cmd} represents the probability of issuing command, *cmd*, given that the history of the system is H_n. A deterministic decision is the special case with $p_{cmd} = 1$ for some command, *cmd*. Over a finite time horizon, the decisions taken by the PM are a finite discrete sequence $\delta_{(1)},\ldots,\delta_{(n)}$. We call this sequence a policy π. The policy π is the free variable of our optimization problem. If a policy $\pi = (\delta_{(1)},\ldots,\delta_{(n)})$ is adopted, then we can define $P_n^{\pi} = \prod_{i=1}^{n} P_{\delta_{(i)}}$, which is simply the n-step transition matrix under policy π.

Example: Consider the example of the previous section. Suppose that the PM observes the following history: s1 = (0, *ON*, 0), s2 = (1, *OFF*, 0) (states in periods 1, 2), and *cmd*(1) = *Go2Sleep* (action taken at time 1.) A possible decision at time 2 in a randomized policy, when state s2 is observed, consists of setting probabilities for issuing commands *Go2Active* and *Go2Sleep* to $p_{Go2Active} = 0.34$, $p_{Go2Sleep} = 0.66$, respectively. On the other hand, in case of a deterministic policy, for example, the PM will decide to issue the *Go2Active* command to the underlying PMC. ♦

Policy Optimization: The problem is to find the optimal policy (set of state-action pairs) for the PM such that some power-related cost function is minimized subject to a set of performance constraints. Consider that the system is in state $s = (s_p, s_q, s_r)$ at the beginning of time period *n*. A typical example of a cost function is the *expected power consumption* of the SP in that time period, which is denoted as $c(s_p, \delta_{(n)})$ and represents the power consumption when the SP starts in state s_p and the PM takes decision $\delta_{(n)}$. (Note that $c(s_p, \delta_{(n)}) = \sum_{cmd \in \delta_{(n)}} p_{cmd} \cdot c(s_p, cmd)$.) A second parameter of interest is the *performance penalty* in that time period, which is denoted by $d(s_q)$ and is typically set to the queue length, s_q. Finally, one can consider the *request loss* in the time period, denoted by $b(s_r, s_q)$. The loss factor is in general set to one when a request arrives (s_r = 1) and the queue buffer is full ($s_q = S_q$); otherwise it is set to zero. We are interested in finding the optimum stationary PM policy for the system. This means that decision, δ_n, is only a function of the system state, s, and not of the time period at which the decision is made. In other words, the policy sought is one in which the same decision is made for a given global state regardless of time. With this in mind, we can now define a power consumption vector, $\underline{c}(\delta_s)$, a performance penalty vector, $\underline{d}(\delta_s)$, and a request loss vector, $\underline{b}(\delta_s)$. Each vector has $|S|$ elements. Furthermore, the s^th element of each vector is the expected value of the corresponding quantity when the system is in global state *s* and decision δ_n is taken. When performing policy optimization, we want to minimize the expected power consumption while keeping the average performance and request loss below some levels specified by the user. Given the probability distribution of the initial system state at the beginning of the first period, p_1, the problem of determining the optimal stationary policy can be formally described as follows:

$$\begin{aligned} &\min_{\pi} \lim_{N\to\infty} \frac{1}{N} \sum_{n=1}^{N} p_1 \cdot P_{n-1}^{\pi} \cdot \underline{c}(\delta_{(n)}) \\ &\text{s.t.} \\ &\lim_{N\to\infty} \frac{1}{N} \sum_{n=1}^{N} p_1 \cdot P_{n-1}^{\pi} \cdot \underline{d}(\delta_{(n)}) \le D_M \\ &\lim_{N\to\infty} \frac{1}{N} \sum_{n=1}^{N} p_1 \cdot P_{n-1}^{\pi} \cdot \underline{b}(\delta_{(n)}) \le B_M \end{aligned} \tag{3}$$

where $\underline{c}(\delta_{(n)}) = \underline{c}(\delta_s)$ if $s = s_{(n)}$ and D_M and B_M denote the upper bounds on average required performance penalty and request loss in any time period, respectively. The optimization is carried over the set of all possible policies. Hence, solving the aforementioned optimization appears to be a formidable task. Fortunately, if the delay constraint for the EMC is an active constraint, the optimal power management policy will generally be a randomized policy [23]. The randomized optimal policy can be obtained by solving a linear programming problem as explained in [9]. This approach offers significant improvements over previous power management techniques in terms of its theoretical framework for modeling and optimizing the EMC.

There are, however, some shortcomings. First, because the EMC is modeled in the discrete-time domain, some assumptions about the PMCs may not hold for real applications, such as the assumption that each event comes at the beginning of a time slice, or the assumption that the transition of the SQ is independent of the transition of the SP, etc. Second, the state transition probability of the system model cannot be obtained accurately. For example, the discrete-time model cannot distinguish the busy state and the idle state because the transitions between these two states are instantaneous. However, the transition probabilities of the SP when it is in these two states are different. Moreover, the PM needs to send control signals to the PMCs in every time-slice, which results in heavy signal traffic and a heavy load on the system resources (and, therefore, more power.)

Reference [10] overcomes these shortcomings by introducing a new system model based on *continuous-time Markov decision processes* (CTMDP.) As a result of this model, the power management policy becomes asynchronous which is more appropriate for implementation as part of the operating system. The new model considers the correlation between the state of the SQ and the state of the SP, which is more realistic than previous models. Moreover, the service requester model is capable of capturing complex workload characteristics and the overall system model is constructed exactly and efficiently from those of the component models. An analytical based approach is used to calculate the generator matrix for the joint process of SP-SQ and a tensor sum-based method is utilized to calculate the generator matrix of the joint process of SP-SQ and SR. The policy optimization problem under the CTMDP model can be solved using (exact) linear programming and (heuristic) policy iteration algorithms. Moreover, this work models the service queue as two queues consisting of *low-priority* and *high-priority* service requests, which furthermore captures the behavior of real-life EMCs.

Because the CTMDP policy is a randomized policy, at times it may not turn off the SP even when there is no request in the SQ. If the stochastic model exactly represents the system behavior, then this policy is optimal. However, in practice, because the stochastic model is not accurate enough, the CTMDP policy may cause unnecessary energy dissipation by not turning off the SP. For example, the real requests pattern on the SP may be quite different from what has been assumed in theory, and the SP idle time may be much longer than one would expect based on the assumption of exponential input inter-arrival time. In this case keeping the SP on while it is idle can result in power waste. The authors of [10] thus present an improved CTMDP policy (called CTMDP-Poll) by adding a *polling state*. The functionality of the polling state is very simple. After adding this state, even if the CTMDP policy allows the SQ to stay on when the SQ is empty, the policy will re-evaluate this decision after some random-length period of time. For example, if the SQ is empty and the PM has made a decision (with probability of 0.1) of letting the SQ to stay ON, then after 2s, if there is no change in the SQ, the models will enter the polling state, and the PM will have to re-evaluate its decision. At this time, the probability for it to still let SQ remain on is again 0.1. So as the time goes on, the total probability of the SQ remaining in the ON state reduces in a geometric manner. In this way, one can make sure that the SP will not be idle for too long, resulting in less wasteful energy dissipation.

The timeout policy is an industry standard that has been widely supported by many real systems. A DPM technique based on timeout policies may thus be easier and safer for users to implement. At the same time, it helps them achieve a reasonably good energy-performance trade-off. To implement a more elaborate DPM technique requires the users to directly control the power-down and wake-up sequences of system components, which normally necessitates detailed knowledge of hardware and involves a large amount of low-level programming dealing with the hardware interface and device drivers. Notice also that the various system modules typically interact with each other implying that sudden power-down of a system module may cause the whole system to malfunction or become unstable i.e., direct control over the state of a system module is a big responsibility that should not be delegated unceremoniously. A DPM technique based on a simple and well-tested timeout policy and incorporated in the operating system will have none of the above concerns. Based on these reasons, the authors of [24] present a timeout-based DPM technique, which is constructed based on the theory of Markovian processes and is capable of determining the optimal timeout values for an electronic system with multiple power-saving states. More precisely, a Markovian process based stochastic model is described to capture the power management behavior of an electronic system under the control of a timeout policy. Perturbation analysis is used to construct an offline gradient-based approach to determine the set of optimal timeout values. Finally, online implementation of this approach is also discussed.

B.3 Petri Net-Based Approaches

The DPM approaches based on Markov decision processes offer significant improvements over heuristic power management policies in terms of the theoretical framework and ability to apply strong mathematical optimization techniques [25]-[31]. However, previous works based on Markov decision processes only describe modeling and policy optimization techniques for a simple power managed system. Such a system contains one SP that provides services (e.g. computing, file access, etc.), one SQ that buffers the service requests for the SP, and one SR that generates the requests for SP. It is relatively easy to construct the stochastic models of the individual components because their behavior is rather simple. However a significant effort is required to construct the joint model of SP and SQ mostly because of the required synchronization between state transitions of SP and SQ. Furthermore, the size of the Markov process model of the overall system rises rapidly as the number of SPs and SRs is increased.

Generalized Stochastic Petri Nets (GSPNs) target more complex power-managed systems as shown in Figure 8. The example depicts a typical multi-server (distributed computing) system. Note that this model only captures those system behaviors that are related to the power management. The system contains multiple SPs with their own Local SQs (LSQ.) There is a SR that generates the tasks (requests) that need to be serviced. The Request Dispatcher (RD) makes decisions about which SP should service which request. Different SPs may have different power/performance parameters. In real applications, the RD and LSQs can be part of the operating system, while SPs can be multiple processors in a multi-processor computing system or number of networked computers of a distributed computing system.

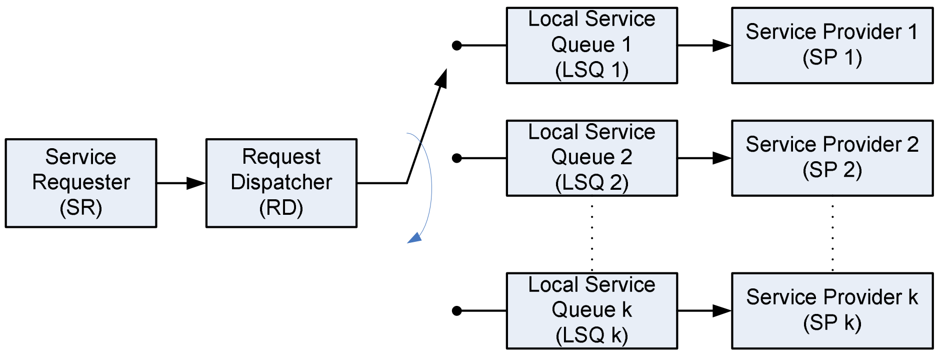

Service Requester (SR)
Request Dispatcher (RD)
Local Service Queue 1 (LSQ 1)
Service Provider 1 (SP 1)
Local Service Queue 2 (LSQ 2)
Service Provider 2 (SP 2)
Local Service Queue k (LSQ k)
Service Provider k (SP k)

Figure 8. A multi-server/distributed-computing system.

The complexity of the modeling problem for the above system is high not only because of the increased number of components, but also because of the complex system behaviors that are present. For example, one needs to consider the synchronization of LSQs and SPs, the synchronization of the SR and LSQs, the dispatch behavior of the RD, and so on. In this situation when complex behaviors must be captured by the system model, the modeling techniques in [11] become inefficient because they only offer stochastic models for individual components and require that global system behaviors be captured manually. Obviously, we need new DPM modeling techniques for large systems with complex behaviors.

For a detailed introduction to Petri Nets, please refer to [33].

A *Petri Net* (PN) model is graphically represented by a directed bipartite graph in which the two types of nodes (places and transitions) are drawn as circles, and either bars or boxes, respectively (cf. Figure 9.) The edges of the graph are classified (with respect to transitions) as:

- input edges: arrow-headed edges from places to transitions
- output edges: arrow-headed edges from transitions to places

Multiple (input/output) edges between places and transitions are permitted and annotated with a number specifying their multiplicities.

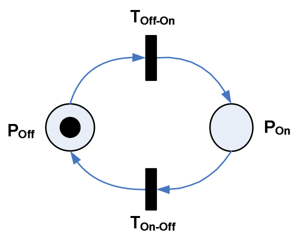


Figure 9. PN description of a switch

Places can contain *Tokens*, which are drawn as black dots within places. The state of a PN is called *marking*, and is defined by the number of tokens in each place. As in classical automata theory, in PN there is a notion of initial state (*initial marking*.) Places are used to describe possible local system states (named conditions or situations.) Transitions are used to describe events that may modify the system state. Edges specify the relation between local states and events in two ways: they indicate the local state in which the event can occur, and the local state transformations induced by the event.

The dynamic behavior of the PN is governed by the firing rule. A transition can fire (an event takes place) if all the transition input places (i.e., those places connected to the transition with an arc whose direction is from the place to the transition), contain at least one token. In this case the transition is said to be enabled. The firing of an enabled transition removes one token from all of its input places, and generates one token in each of its output places (i.e., those places connected to the transition with an arc whose direction is from the transition to the place.) The firing of a transition is an atomic operation. Tokens are removed from input places, and deposited into output places with one indivisible action. Typically, the firing of a transition describes the result of either a logical condition becoming true in the system, or the completion of an activity. The latter interpretation is the reason for associating timing with transitions, as many authors did in their proposals for the definition of temporal concepts in PNs. Hence, *time* can be naturally associated with transitions. In the semantics of PNs, this type of transitions with associated temporal specifications is called a *timed transition*. These transitions are represented graphically by boxes or thick bars and are denoted with names that start with T. On the other hand, *Immediate transitions* fire as soon as they become enabled (with zero delay), thus acquiring a sort of precedence over timed transitions, and leading to the choice of giving priority to immediate transitions in the definition of GSPNs. In this chapter, immediate transitions are depicted as thin bars.

It should be noted that the PN state transformation is local, in the sense that it involves only the places connected to a transition by input and/or output arcs (this will be visible in the forthcoming examples; the PN model of a switch is so simple that local and global states coincide.) This is one of the key features of PNs, which allows compact description of distributed systems.

Example: A simple example of a PN model is given in Figure 9, where two places P_{ON} and P_{OFF} , and two transitions T_{ON-OFF} and T_{OFF-ON} are connected with four arcs. Both places define conditions (i.e., the "ON condition" or the "OFF condition".) The state depicted in the Figure 9 is such that place P_{OFF} contains one token; thus the "OFF condition" is true; instead, since place P_{ON} is empty, the "ON condition" is false. In this simple example, transition T_{OFF-ON} is enabled, and it fires, removing one token from P_{OFF} and depositing one token in P_{ON}. The new state is such that the "ON condition" is true and the "OFF condition" is false. In the new state, transition T_{ON-OFF} is enabled, and it fires restoring the state shown in Figure 9. The simple PN model in Figure 9 may be interpreted as the PN description of the behavior of a switch. ♦

In a *Stochastic Petri Net* (SPN) model, each timed transition is associated not only with a single transition time but a collection of randomly generated transition times from an exponential distribution. As in case of timed transitions, for the description of SPNs, one can assume that each timed transition possesses a timer. When the transition becomes enabled for the first time after firing, the timer is set to a value that is sampled from the exponential pdf associated with the transition. During all time periods in which the transition is enabled, the timer is decremented. Transitions fire when their timer readout goes down to zero. With this interpretation, each timed transition can be used to model the execution of an activity in a distributed environment; all activities execute in parallel (unless otherwise specified by the PN structure) until they complete. At completion, activities induce a local change of the system state, which is specified with the interconnection of the transition to input and output places. In Generalized Stochastic Petri Nets (GSPNs), the exponentially timed and immediate transitions coexist in the same model. In the context of GSPNs places can be divided into two different classes based on the type of the transitions for which they are inputs, i.e., *vanishing places* and *tangible places*. The place is a vanishing place if it is the only input place of an immediate transition; otherwise the place is called a tangible place.

Let's see how one can use this modeling tool to develop a DPM policy by capturing the exact behavior of a complex EMC system. To capture the energy consumption of the EMC in the GSPN model, the following two definitions are necessary.

Definition: A GSPN with cost is a GSPN model with the addition of two types of cost: *impulse cost* associate with marking transitions and *rate cost* associated with places. Impulse cost occurs when the GSPN makes a transition from one marking to another. Rate cost is the cost per unit time when the GSPN stays in a certain marking.

Definition: A controllable GSPN with cost is a GSPN where all or part of the probabilities of timed transitions can be controlled by outside commands.

***Example*:** Consider that a SP in the processor has two power states: {*ON*, *OFF*}. In the *ON* state, the SP provides service with an average service time of 5ms. The average time to switch from the *ON* state to *OFF* state is 0.66ms, the average time to switch from the *OFF* state to *ON* state is 6ms. The power consumption of the SP is 2.3W when it is in the *ON* state and 0.1W when it is in the *OFF* state. The energy needed to switch from the *ON* state to *OFF* state is 2mJ, the energy needed to switch from the *OFF* state to *ON* state is 30mJ. Assume that the maximum length of the SQ is 3. Figure 10 shows the GSPN model of the single processor system. The input gate

$G_{capacity}$ sets the SQ capacity constraint. The place $P_{ON\text{-}OFF}$ denotes the SP status when it is switching from the *ON* state to *OFF* state while the place $P_{OFF\text{-}ON}$ denotes the SP status when it is switching from the *OFF* state to *ON* state. The place $P_{idle(ON)}/P_{idle(OFF)}$ denotes the SP status when it is idle and the power state is *ON*/*OFF*. The place $P_{work(ON)}$ denotes the SP status when it is working and the power state is *ON*. The SP will have exactly one such status at any time. From the topology of the GSPN, one realizes that the sum of tokens in places $P_{ON\text{-}OFF}$, $P_{idle(ON)}$, $P_{work(ON)}$, $P_{idle(OFF)}$, $P_{OFF\text{-}ON}$ is 1 at any time.

The number of tokens in P_{SQ} denotes the number of waiting requests in the SQ. The initial marking of $P_{idle(ON)}$ is 1 while the initial marking of the other places is 0, which indicates that the initial state of the SP is idle and the initial state of SQ is empty. The places $P_{decision(ON)}$, $P_{decision(OFF)}$ are vanishing places. They indicate the very short period of time when the SP is taking command from PM and is in the *ON* or *OFF* state. The place $P_{changing}$ is also a vanishing place. It is an auxiliary place, which indicates that the state of the system is changing so that it is time for the SP to receive the power management command if it is currently idle. $T_{ON\text{-}OFF}$ and $T_{OFF\text{-}ON}$ are timed activities. They indicate the time needed to switch from the *ON* state to *OFF* state and the time needed to switch from the *OFF* state to *ON* state. $T_{processing}$ is also a timed transition, which indicates the time needed to process one request. T_{input} denotes the time needed to generate the next request. It actually belongs to the GSPN model of the request generation system. $T_{decision(ON)}$ and $T_{decision(OFF)}$ are immediate transitions. They represent the process of randomized action issued by the power manager (PM.) The two cases in $T_{decision(ON)}$ or $T_{decision(OFF)}$ are mutually exclusive. The case probability equals the action probability, which is marking and policy dependent. If the policy is unknown, the GSPN is a controllable GSPN. When the SP is idle and *ON* (a token is in place $P_{idle(ON)}$) and SQ is not empty, the immediate transition T_{start} is completed which indicates that the SP enters the busy state. When the SP is *OFF* (a token is in place $P_{idle(OFF)}$) and the state of SQ is changing (a token is in place $P_{changing}$), the immediate transition $T_{re\text{-}decision}$ is completed which indicates that the SP returns to the action taking stage. If the SP is not *OFF* and the state of SQ is changing, the immediate transition T_{vanish} is completed which indicates that the change is ignored.

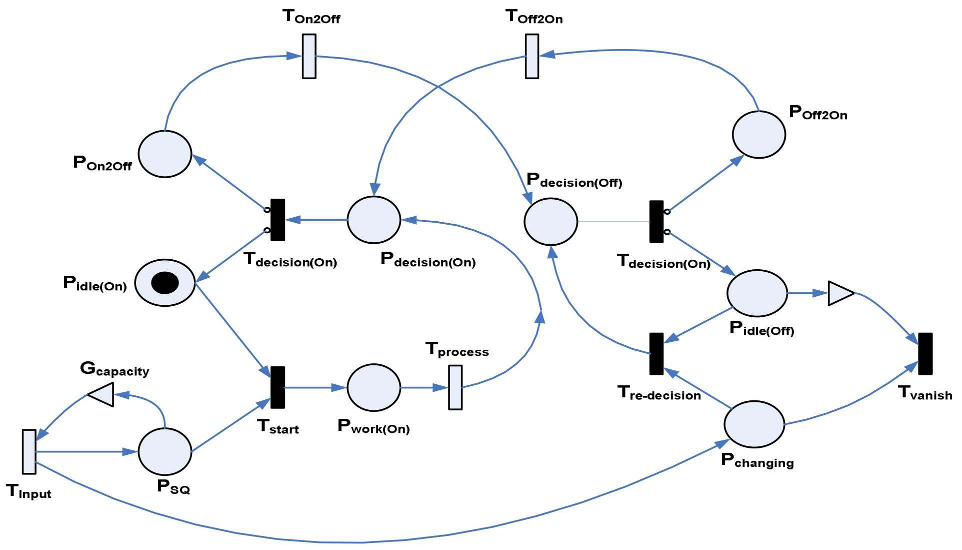


Figure 10. Example GSPN model of a Single Requester-Single Server system.

In the complex system the *Request Generation System* (RGS) can be very complicated and cumbersome. RGS can generate various types of requests. The generation time of different types of request may be different. Some types of requests can be serviced by several different SPs; whereas some other types of requests can be serviced by only a certain SP. There may exist correlations among the generation of different types of requests. If the SQ is full, the RGS will stop generating request. It will resume request generation when there is vacancy in the SQ.

Example: Assume that there are three types of requests, one is type A which can only be serviced by SP A, one is type B which can only be service by SP B, the other is type AB which can be serviced by both SP A and SP B. The correlations among these requests are given by a probabilistic matrix, For example, from the matrix one will know that the probability that a type AB request was issued after a type A request is 0.6. Figure 11 shows the GSPN model of this RSG. In this figure, the case probability of activities $T_{switch(A)}$, $T_{switch(B)}$ and $T_{switch(AB)}$ takes value from a probabilistic matrix. The input gate G_{cap_A} represents the condition that the SQ in SP A is not full. The input gate G_{cap_B} represents the condition that the SQ in SP B is not full. The input gate G_{cap_AB} represents the condition that the SQ in SP A or the SQ in SP B is not full. Notice that the input places of these input gates belong to the GSPN model of each SP. These input gates enable or disable the request generation. For example, if the condition given by G_{cap_A} is false, which means that SQ of SP A is full, then the time activity $T_{gen(A)}$ is disabled, which means that request generation procedure of type A request pauses. $T_{gen(A)}$ will be enabled when the condition given by G_{cap_A} becomes true, which means that the request generation procedure resumes when there is a vacancy in the SQ. ♦

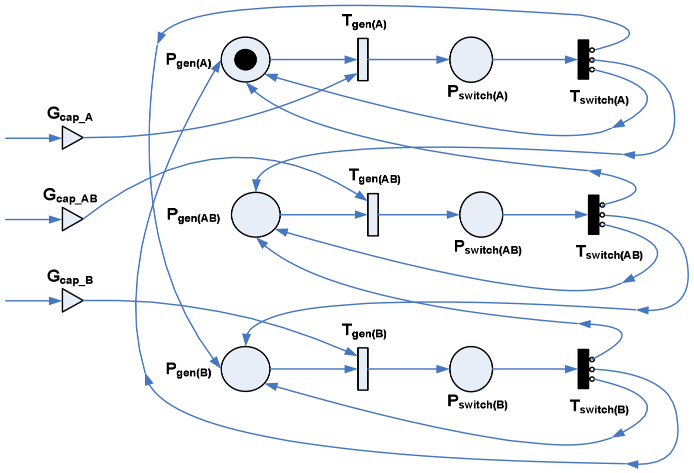


Figure 11. Example GSPN model for an RGS.

To model a complex system composed of several SP's, similar to the one shown in Figure 8 with an RGS and interactions among the components, a hierarchical approach can be used. First, each SP is modeled using a single requester–single server model. Next, the requests generated by the RGS are sent to the SP #i with probability p_i through a dispatcher. If the request can only be serviced by SP #i, then p_i is 1. If the request cannot be serviced by SP #i then p_i is 0. In all other cases the probability p_i is controlled by the dispatcher. The optimal dispatch policy can be obtained by solving a Markov decision process. The GSPN model of such complex system contains the following components:

1. The GSPN models of RGS and SP's.
2. A set of input gates $\{G_{cap\_i}\}$. The input place of a G_{cap_i} is the P_{SQ} of all SP's, which can provide service for request type i. The activity of G_{cap_i} is $T_{gen(i)}$. A gate G_{cap_i} indicates the condition that there are free positions in SQ to buffer the request.

3. Arcs from transition $T_{gen(i)}$ in RGS to place P_{SQ} in any SP that can provide service for request *i*.

Example**:** Consider a complex system which contains two SP's and one RGS as described in a previous example.

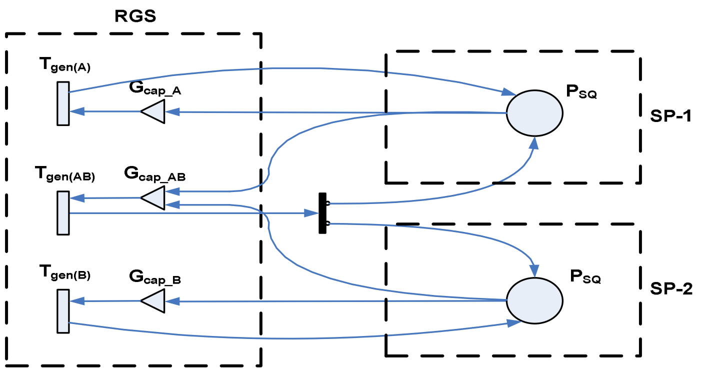


Figure 12. Example GSPN model of a complex system with two SP's and three request types.

Figure 12 shows the GSPN model of this system. ♦

After generating the controllable GSPN model, one can reduce it to a SPN, which is the same as a GSPN except that SPN does not have instantaneous activities [32]. From the SPN, we can find its reachability graph, and thereby, generate the corresponding continuous time Markov decision process. The state s_i of the CTMDP corresponding to the marking M_i in the reachability set. The rate cost of s_i is the sum of the rate costs of places in M_i. The transition cost of the CTMDP from state s_i to s_j is the impulse costs of the completed activities when the GSPN is switching from marking M_i to M_j. The reader may refer to [32] for the procedure of reducing a GSPN to a SPN. The optimal policy is CTMDP is obtained by solving a set of linear programming. A GSPN can be converted to a CTMDP, hence it can be evaluated efficiently. However, the exponential distribution is not always an appropriate way to represent the transition time. If the transition time has a general distribution, the Markovian property will be destroyed. This problem can be circumvented by using the stage method [31], which approximates the general distributions using the series or parallel combinations of exponential stages.

Conclusions

This chapter described various dynamic power management approaches for performing energy efficient computation: predictive shutdown, Markovian decision process-based, and generalized stochastic Petri net-based approaches. A significant reduction in power consumption can be obtained by employing these DPM techniques. For example, for applications where continuous computation is not being performed, an aggressive shut down strategy based on an online predictive technique can reduce the power consumption by a large factor compared to the straightforward conventional schemes where the power down decision in based solely on a predetermined idle time threshold. Moreover, predictive shutdown heuristic may be applied to manage the shutdown of peripherals such as disks. An on-line algorithm that makes the shutdown decision using a prediction of the time to next disk access can result in higher power reduction compared to more conventional threshold based policies for disk shutdown.

On the other hand, construction of optimal power management policies for low-power system is a critical issue that cannot be addressed by using common sense and heuristic solutions such as those used in predictive shutdown schemes. Stochastic models provide a mathematical framework for the formulation of power-managed devices and workloads. The constrained policy optimization problem can be solved exactly in this modeling framework. Policy optimization can be cast into a linear programming problem and solved in polynomial time by efficient interior point algorithms. Moreover, tradeoff curves of power versus performance can be computed. Furthermore, adaptive algorithms can compute optimal policies in systems where workloads are highly non-stationary and the service provider model changes over time. CTMDP-based techniques introduce a new and more complete model of the system components, as well as the model of the whole system. This mathematical framework captures the characteristics of the real applications more accurately which is mainly because the problem is solved in continuous-time domain while previous approaches solve the problem in discrete-time domain.

A shortcoming of DTMDP or CTMDP-based techniques is that it is very difficult to use these modeling frameworks when attempting to represent complex systems, which in turn consist of multiple closely interacting SPs and must cope with complicated synchronization schemes. In this case, generalized stochastic Petri Nets (GSPN) and the corresponding modeling techniques based on the theory of GSPN have proven to be quite effective. The constructed GSPN model can be automatically converted to an isomorphic continuous-time Markov decision process. From the corresponding Markov decision process, one can calculate the optimal DPM policy, which achieves minimum power consumption for given delay constraints. In real applications, the inter-arrival time of service requests may not follow an exponential distribution, for example, thety could have heavy-tail distributions such as Pareto distribution. This problem can be solved by using the “stage method” (i.e., approximating the given source of requests by a series-parallel connection of exponentially distributed sources.)

Acknowledgement: This work was sponsored in part by DARPA and NSF CSN program office.

Bibliography

[1] S. Gary et al., “PowerPC 603, a microprocessor for portable computers,” IEEE Design & Test of Computers, vol. 11, pp. 14–23, 1994.

[2] “Advanced micro devices,” in AM29SLxxx Low-Voltage Flash Memories, 1998.

[3] E. Harris et al., “Technology directions for portable computers,” Proc. IEEE, vol. 83, pp. 636–657, Apr. 1996.

[4] 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.

[5] H. Shim, N. Chang, and M. Pedram, "A backlight power management framework for the battery-operated multi-media systems," IEEE Design and Test of Computers, Vol. 21, No. 5, Sept./Oct. 2004, pp. 388-396.

[6] SA-1100 Microprocessor Technical Reference Manual, Intel, 1998.

[7] M. Srivastava, A. Chandrakasan, and R. Brodersen, "Predictive system shutdown and other architectural techniques for energy efficient programmable computation," IEEE Trans. VLSI Systems, Vol. 4, pp. 42–55, Mar. 1996.

[8] C-H. Hwang and A. Wu, “A predictive system shutdown method for energy saving of event-driven computation,” Proc. of Int'l Conf. on Computer-Aided Design, pp. 28–32, Nov. 1997.

[9] 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, Jun. 1999.

[10] Q. Qiu, Q Wu and M. Pedram, "Stochastic modeling of a power-managed system-construction and optimization," IEEE Trans. Computer-Aided Design, Vol. 20, pp. 1200-1217, Oct. 2001.

[11] T. Simunic, L. Benini, P. Glynn, G. De Micheli, "Event-driven power management," IEEE Trans. Computer-Aided Design, Vol. 20, pp.840-857, Jul. 2001.

[12] E.-Y. Chung, L. Benini, and G. D. Micheli., ”Dynamic power management using adaptive learning trees.,” In Proceedings of ICCAD, 1999.

[13] Y. Lu and G. DeMicheli., ”Adaptive hard disk power management on personal computers.,” In Proceedings of the Great Lakes Symposium on VLSI, 1999.

[14] S. Irani, R. Gupta, and S. Shukla. “Competitive analysis of dynamic power management strategies for systems with multiple power savings states,” *IEEE Conference on Design, Automation and Test in Europe*, 2002.

[15] S. Irani, S. Shukla, and R. Gupta. “Online strategies for dynamic power management in systems with multiple power saving states,” *IEEE Trans. on Embedded Computing Systems*, 2003.

[16] P. Krishnan, P. Long, and J. Vitter, “Adaptive disk spin-down via optimal rent-to-buy in probabilistic environments,” in Int. Conf. Machine Learning, July 1995, pp. 322–330.

[17] 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.

[18] 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.

[19] Y. Lu and G. De Micheli, “Adaptive hard disk power management on personal computers,” in Great Lakes Symp. VLSI, Feb. 1999, pp. 50–53.

[20] A Karlin, M. Manasse, L. McGeoch, and S. Owicki. “Randomized competitive algorithms for non-uniform problems,” *ACM-SIAM Symposium on Discrete Algorithms*, pages 301–309, 1990.

[21] D. Gross and C. M. Harris, Fundamentals of Queuing Theory, Wiley (1985).

[22] P. Rong and M. Pedram, “Battery-aware power management based on Markovian decision processes,” Proc. of Int'l Conference on Computer Aided Design, Nov. 2002, pp. 712-717.

[23] E. V. Denardo, “On Linear Programming in a Markov Decision Problem,” Management Science, Vol. 16, No. 5, pp. 281-288, January, 1970.

[24] P Rong and M. Pedram, “Determining the Optimal Timeout Values for a Power-Managed System based on the Theory of Markovian Processes: Offline and Online Algorithms,” Proc. of Design Automation and Test in Europe, 2006.

[25] U. Narayan Bhat, “Elements Of Applied Stochastic Processes,” John Wiley & Sons, Inc. 1984

[26] B. Miller, “Finite State Continuous Time Markov Decision Processes With an Finite Planning Horizon.” SIAM J. Control, Vol. 5, No. 2, pp. 266-281, 1968.

[27] B. Miller, "Finite State Continuous Time Markov Decision Processes With an Infinite Planning Horizon". J. Of Mathematical Analysis and Applications, No. 22, pp. 552-569, 1968.
[28] R.A. Howard, Dynamic Programming and Markov Processes, Wiley, New York, 1960
[29] D. P. Heyman, M. J. Sobel, Stochastic Models in Operations Research, McGraw-Hill Book Company, 1982
[30] G. Bolch, S. Greiner, H. D. Meer and K. S. Trivedi, Queuing Networks and Markov Chains, John Wiley & Sons, Inc., 1998
[31] L. Kleinrock, Queuing Systems. Volume I: Theory, Wiley-Interscience, New York, 1981.
[32] M. A. Marsan, G. Balbo, G. Conte, S. Donatelli and G. Franceschinis, Modeling With Generalized Stochastic Petri Nets, John Wiley & Sons, New York, 1995.
[33] Jiacun Wang, "Timed Petri Nets: Theory and Application", Springer – Mathematics, 1998
[34] Z. Ren, B. H. Krogh, R. Marculescu, "Hierarchical Adaptive Dynamic Power Management," February 2004 Proceedings of the conference on Design, automation and test in Europe - Volume 1.