To appear in *Energy Efficient Distributed Computing Systems*, Editors: A. Zomaya and Y-C Lee, Wiley-IEEE Computer Society Press, Aug. 2012

# A Stochastic Framework for Hierarchical System-Level Power Management

**Peng Rong[1] and Massoud Pedram[2]**

**[1]Brocade Communications Systems**
**San Jose, CA, U.S.A.**
**[2]University of Southern California**
**Los Angeles, CA, U.S.A.**

*Summary* - **This article presents a hierarchical power management architecture which aims to facilitate power-awareness in an Energy-Managed Computer (EMC) system with multiple components. The proposed architecture decomposes the power management (PM) task into two layers: component-level and system -level. The component-level PM policy is pre-specified and fixed whereas the system-level PM policy optimization, which is formulated as a concurrent service request flow regulation and application scheduling problem, is solved by the hierarchical power manager. Experimental results show that the hierarchical PM achieves a 25% reduction in the total system energy compared to an optimal component-level DPM policy.**

## Contents

## I. INTRODUCTION

Dynamic power management (DPM), which refers to a selective shut-off or slow-down of components that are idle or underutilized, has proven to be a particularly effective way of reducing power dissipation in such systems. In the literature, various DPM techniques have been proposed, from heuristic methods presented in early works [1][2] to stochastic optimization approaches [3][4]. Among the heuristic DPM methods, the time-out policy is the most widely used approach and has been implemented in many operating systems. The time-out policy is simple and easy to implement, but it has many shortcomings, such as not making use of the statistical information about the service request rates, and having a limited ability to trade-off performance and energy dissipation. Stochastic approaches are mathematically rigorous approaches which are based on stochastic models of service requests and are thus able to derive provably optimal DPM policies.

Reference [6] considered job scheduling as part of a power management policy and proposed an on-line scheme that groups jobs based on their device usage requirements and then checks every possible execution sequence of the job groups to find out the one with minimal power consumption. This work is quite valuable because it demonstrates the potential for additional power saving by doing job scheduling. However, this work also has a few shortcomings. First, each time a new job is generated, the search procedure to find the minimal-power execution sequence has to be repeated. Second, this scheme does not explore the possibility of reducing the system energy by changing the working state of devices that have multiple functional states. Third, exact knowledge of the device usage of a job is required before the job can be scheduled. It is also assumed that this device usage profile does not change during the lifetime of a job. It is not clear how this scheme can capture the dependence between two parts of the same job, if the two parts exhibit very different device usage behavior. Finally, this scheme does not make use of any prediction or expectation of the future behavior of the system, and thus, can only make a greedy online decision.

To capture dependencies between different system components, a power manager must have a global view of the system architecture, connection among components, system resources that are shared among these components, and any possible functional dependency between the components. In addition, application-level scheduling requires the power manager to work closely with the operating system scheduler. Both of these tasks are beyond the capabilities of the existing component-level power management solutions.

A number of power saving mechanisms have been already incorporated into various standards and protocols. Examples are the power management function defined in USB bus standard and the power saving mode in the IEEE 802.11 protocol. A USB device will automatically enter a suspend state if there is no bus activity for three milliseconds. A Wireless Local Area Network (WLAN) card operating in the power saving mode, needs to wake up periodically at the beginning of a beacon interval and listen for traffic identification message.

In most cases, these built-in power-management solutions cannot be changed because they ensure the correct functionality of a device running the related protocol. In this sense, we consider such a device as an uncontrollable or self power-managed component. Even beyond protocol considerations, vendors have already begun to develop power management software specifically designed for their products. An example is the enhanced adaptive battery life extender (EABLE) for Hitachi (IBM Storage Systems, originally) disk drive, which is self-managed and is incorporated into the device driver [7]. EABLE dynamically determines the appropriate mode based on the actual disk access pattern and the internal level of drive activity. Finally, implementation of the device power manager by the designers and manufacturers of the device itself, may relieve the system integrators of the burden of mastering detailed hardware and device driver expertise, and thus facilitate power-awareness in system integration with multiple components.

The component designer does not know the global characteristics and performance requirements of the system in which the component will be incorporated. Therefore, the best that she can do is to provide a generic local power management policy for the component but make some tuning parameters of the local policy controllable by the system designer and the system-level power manager. On the other hand, a system engineer, who devises the architecture of an EMC system and takes care of interfacing and synchronization issues among the selected components, can devise a global power management policy that may help local power manager to improve power efficiency of the component.

Based on the above considerations, we define the problem of hierarchical power management for an EMC system with self power-managed components. More specifically, this article targets a uni-processor computer system which consists of multiple I/O devices. It is possible to extend the proposed approach and apply it to a multi-processor system or a computer cluster, a task which is beyond the scope of this article. The problem is then formulated as a mathematical program with the aid of continuous-time Markovian decision processes (CTMDP) models and solved accordingly.

The key contributions of this article may be summarized as follows.

1. A hierarchical DPM architecture is proposed, where the power management function is decomposed into system and component levels. This division facilitates the integration of various power management techniques into a two-tiered organization and enhances system level power awareness. At the system level, flow control on the service request traffic is employed to improve the effectiveness of built-in component-level power management solutions. Note that the proposed power management architecture can easily handle service providers with or without built-in local power managers.
2. CTMDP-based application-level scheduling is incorporated into system-level power management to achieve further power reduction. This scheduling is stochastically optimized by using the CTMDP model. Applications are scheduled based on the global system state comprising the states of the individual components, the number of waiting tasks, and application stochastic characteristics. In this way, our proposed solution is very different from that of [6].
3. The proposed system-level power management handles component state dependencies, whereby the state of a service provider is affected by states of the other service providers.

The remainder of this article is organized as follows. In Section 2, related works are discussed. The background of CTMDP is introduced in Section 3. Details of the proposed hierarchical DPM framework are described in Section 4. In Section 5, stochastic model of the system-level power management is provided. The energy optimization problem is formulated and solved as a mathematical program in Section 6. Experimental results and conclusions are provided in Sections 7 and 8, respectively.

## II. RELATED WORK

The CTMDP based DPM approach was first proposed in reference [4]. CTMDP-based approach makes policy changes in an asynchronous and event-driven manner and thus surmounts the shortcoming of an earlier work based on discrete-time Markovian decision processes [3], which relied on periodical policy evaluation. Therefore, CTMDP-based DPM approach is more suitable for implementation as part of a real-time operating system environment because of its even-driven nature. Due to space limitation, the background for CTMDP models is not provided here. Interested readers may refer to [5].

The researchers in the literature also proposed other stochastic DMP approaches. Reference [8] improved on the modeling technique of [3] by using time-indexed semi-Markovian decision processes. Recently, the authors [9] discussed several promising DPM techniques including partial-observable Markovian decision processes based approach; however, no results are published so far.

In the literature, some works related to the hierarchical power management have been reported. Reference [10] proposes a DPM methodology for networks-on-chips, which combines node and network centric DPM decisions. More specifically, the node centric DPM uses Time-indexed Semi-Markovian decision processes whereas the network centric DPM allows a source node to use network sleep/wakeup requests to force sink nodes to enter specified states. Our proposed work differs from this approach by providing a more general and mathematically rigorous framework for defining and solving hierarchical DPM problem in an EMC system. In particular, application-level scheduling is exploited and component state dependency is considered by the system-level power manager. In addition, by using a globally-controlled service request flow regulation process, our framework can handle self-power-managed service providers and dynamically adjust their local power management policies. Reference [11] proposes a hierarchical scheme for adaptive DPM under non-stationary service requests, where the term “hierarchical” refers to the manner by which the authors construct a DPM policy. This is different from what is proposed in the present proposal.  More precisely, in their work, the authors formulate policy optimization as a problem of seeking an optimal rule that switches policies among a set of pre-computed ones. However, this work assumes that the service providers are fully controllable and have not built-in power management policy. This work differentiates service request generation between “modes” (applications), but application-level scheduling is not considered. In addition, it focuses on developing power management policies for a single device.

Another kind of hierarchical power management schemes incorporate into one platform multiple hardware components with identical or exchangeable functionality but different levels of power and performance. Reference [12] proposes a scheme that equips each mobile node two complementary radios (long-distance high-power vs. short-distance low-power) and uses both radios to participate in contact discovery. This scheme controls the wake-up interval of each radio to trade between energy savings and the performance of message delivery. Reference [13] presents a hierarchical power management architectural that focuses on providing high levels of consistency in a laptop by integrating two additional low-power processors: StrongARM and ATmega. In this article, a dedicated distribution of each application over the processors is pre-designed to evaluate power saving.

## III. A Hierarchical DPM Architecture

In this article, we consider a uni-processor computer system which consists of multiple I/O devices, e.g. hard disk, WLAN card, or USB devices. Batches of applications keep running on the system. When an application is running on the CPU, it may send requests to one or more devices for services. A performance constraint is imposed on the average throughput of the computer system. The constraint is defined as a minimum amount of completed application workloads over a fixed period of time. It is also required that each application gets a proportional (fair) amount of CPU execution time over a long period of time. Our objective is to minimize the energy consumption of the computer system. More precisely, this article focuses on reducing energy consumption of the I/O devices. Saving processor and memory energy is out of the scope of this article. Readers interested in these power components can refer to [15] and [16].

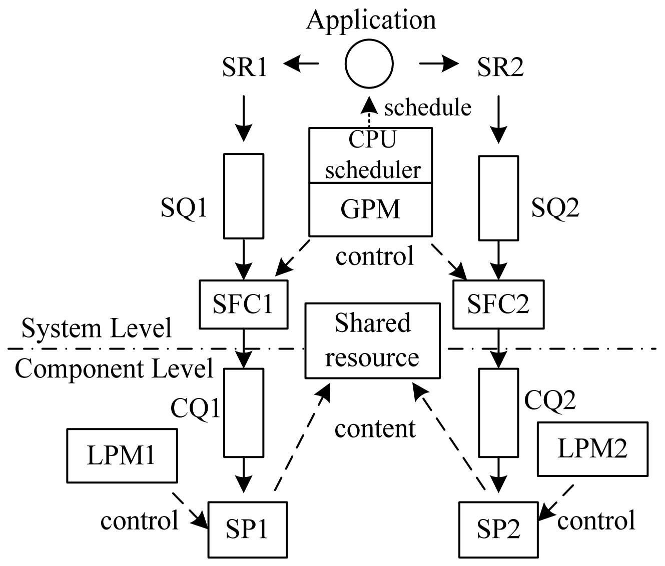

**Fig. 1: Block diagram of a hierarchical DPM structure.**

The architecture of our proposed hierarchical DPM framework which contains two service providers (SP), i.e. two I/O devices, is presented in **Fig.** 1. This architecture has two levels of PM. In the component level, each SP is controlled by a local power manager (LPM). The LPM performs a conventional PM function, i.e., it is monitoring the number of service requests (SR) that are waiting in the component queue (CQ) and consequently adjusts the state of the SP. In the system level, the global power manager (GPM) acts as the central controller which attempts to meet a global performance constraint while reducing the system power consumption. In particular, it performs three separate functions.

1. GPM determines the state of the service flow controller (SFC) and regulates the service request traffic that is subsequently fed into the component queues. Note that in this architecture, the GPM cannot overwrite the LPM policy or directly control the state transition of an SP. Thus, regulating service request flow is the method that the GPM uses to guide the local PM policy and improve the power efficiency of the SPs.
2. The GPM works with the CPU scheduler to select the right applications to run so as to reduce the system power dissipation. This decision is in turn made based on the current state of the PM system, including the states of the SPs and the number of SRs waiting in a service queue (SQ).
3. The GPM resolves the contention for shared resources between different SPs and dynamically assigns the resources so as to increase the system power efficiency. As a side note, the SFC performs three functions, i.e., SR transfer, SR blocking, and fake SR generation, to adjust the statistics of the service request flow that reaches the SP. The SRs that are blocked by the SFC are kept in the SQ.

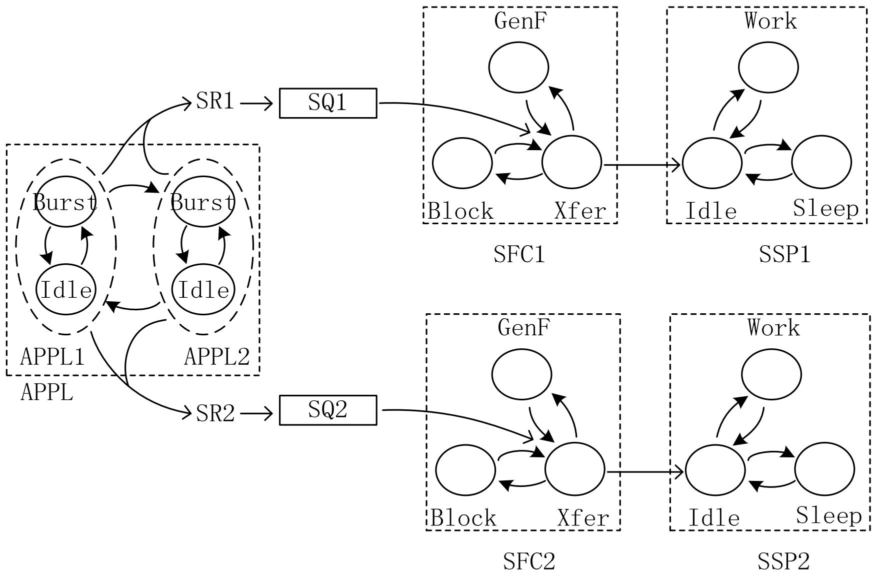

**Fig. 2: CTMDP model of the hierarchical DPM structure.**

## IV. Modeling

We represent the hierarchical DPM structure by a CTMDP model as shown in Fig. 2. This model, which is constructed from the point of view of the GPM, is utilized to derive a system-level PM policy. The CTMDP model contains the following components: an application model (APPL), the SQ, the SFC, and a simulated service provider (SSP).

The SSP is a CTMDP model of the LPM-controlled SP as seen by the GPM. More precisely, it is a composition of the state-transition diagram of the SP and the corresponding LPM policy. Notice that the CQ model is not needed because from the viewpoint of the GPM, the CQ and SQ are viewed as being identical. In the following subsections, the APPL, SFC and SSP models are described in detail followed by modeling of the dependencies between the SPs. An example transition diagram for the SSP is provided in Fig. 2.

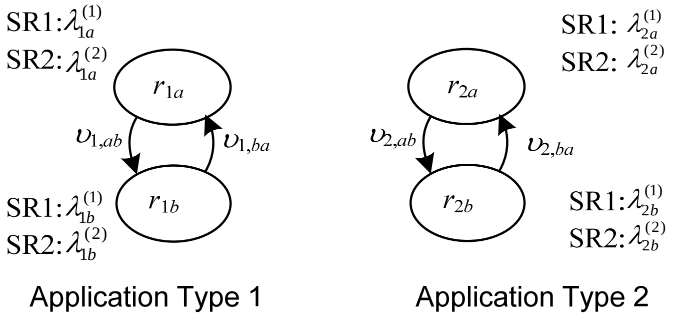

**Fig. 3: CTMDP models of application types 1 and 2.**

### *A. Model of the Application Pool*

It is assumed that the applications running on the computer system can be classified into different types based on their *workload characteristics*, i.e., their SR generation rates and the target SPs (i.e., service destinations.) In reference [11], the authors report that the pattern of SRs that are generated by an application and sent to a hard disk may be modeled by a Poisson process. Here, we use a more general model, i.e., a CTMDP model to describe the complex nature of SR generation of an application. When an application that is running on the CPU moves from one internal state to next, it generates various types of SRs with different rates. For example, as illustrated in Fig. 3, in state $r_{1a}$, application type 1 generates SR1 with a rate of $\lambda_{1a}^{(1)}$ and SR2 with a rate of $\lambda_{1a}^{(2)}$ . Similarly, in state $r1_b$, the generation rates for these two SRs become $\lambda_{1b}^{(1)}$ and $\lambda_{1b}^{(2)}$ , respectively. In state $r_{1a}$, application type 1 transits to state $r_{1b}$ with a rate of $\upsilon_{1,ab}$, which also implies that the average time for application type 1 to stay in state $r_{1a}$ is $1/\upsilon_{1,ab}$.

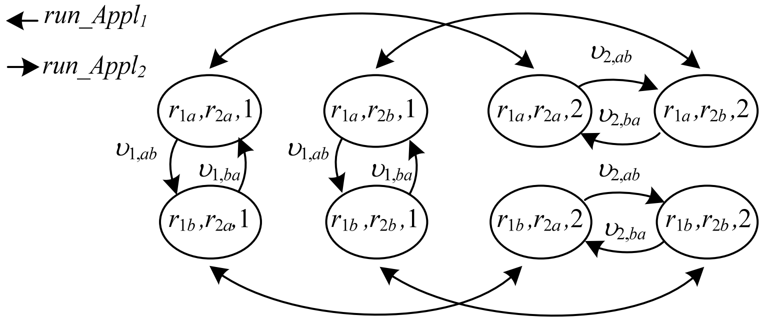

**Fig. 4: CTMDP model of an application pool.**

By using the CTMDP model for each application type, we can set up the CTMDP model of an *application pool*, $S_{APPL}$. A state of $S_{APPL}$ is a tuple comprising of the corresponding state for every application type plus information about the application currently running on the CPU. The CTMDP model of the example $S_{APPL}$ , as depicted in Fig. 4, h eight global states, ($r_{1x}$,$r_{2y}$,*flag*) where $r_{1x}$ denotes the service generation state *x* for application 1 while $r_{2y}$ denotes state y for application type 2. *flag*=1 (2) means that the first (second) application is running. For example, ($r_{1a}$,$r_{2a}$,1) means that application type 1 is running and it is in state $r1_a$. Furthermore, the state of application type 2 was $r_{2a}$ just before it was swapped out. The CTMDP model has a set of autonomous transitions between state pairs with the same activation flag value. The transition rates are denoted by $\upsilon_{i,xy}$ where *x* and *y* denotes the service generation states of application type *i*. For example, the transition between ($r_{1a}$,$r_{2a}$,1) and ($r_{1b}$,$r_{2a}$,1) is autonomous. Notice that a transition from ($r_{1a}$,$r_{2a}$,1) and ($r_{1b}$,$r_{2b}$,1) is disallowed because application 2 is not running therefore, it cannot possibly change its service generation state. The model also has a set of action-controlled transitions between global states with the same $r_{1x}$,$r_{2y}$ values.

The action set is $A_{APPL}$ = {*run_Appl<sub>i</sub>*}, where $Appl_i$ denotes application type *i*. For example, if the global state of the $S_{APPL}$ is ($r_{1x}$,$r_{2y}$,1) and action *run_Appl2* is issued then the new global state of the system will be ($r_{1x}$,$r_{2y}$,2). A transition between ($r_{1a}$,$r_{2b}$,1) and ($r_{1a}$,$r_{2a}$,2) in not allowed because it implies that during context switch from application type 1 to type 2, the service generate state of application 2 changed, an impossibility in our model.

The number of states grows exponentially with the number of application types. Thus, to mitigate scalability issue, one must group all interesting applications into a relatively small number of application types. According to our experimental results and observations, although the number of different applications may be large, the number of different application classes is rather small.

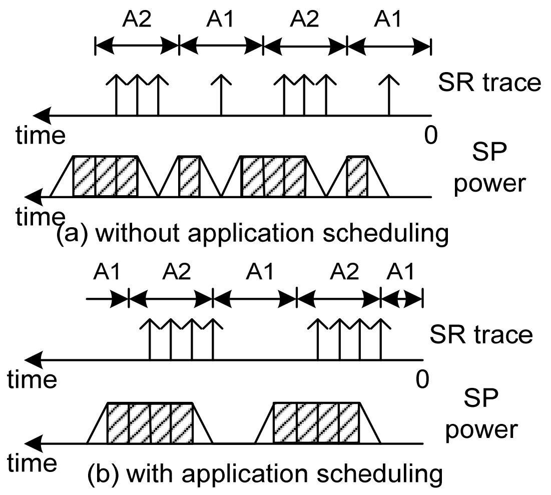

**Fig. 5: An example of the effectiveness of application scheduling.**

The reason that application scheduling based on the global system state can reduce the total system power consumption can be explained by a simple example. Let's consider a system with only one SP. There are two application types A1 and A2. A1 generates SRs at a rate of 1 request per unit time while A2 generates 3 requests per unit time. The SP wakes up as soon as a request is generated and sleeps when all requests have been serviced. Two execution sequences will be considered. In the first sequence, there is no application scheduling. Each application is alternately executed for exactly one unit of time. In the second sequence, we perform application scheduling based on the number of waiting requests in the SQ. More precisely, during the running period of A1, as soon as a request is generated, the scheduler switches to A2. After A2 is run for one unit of time, A1 will be brought back to continue its execution. This policy ensures that all SRs that are targeted to the SP are bundled together and that the SP sleep time is maximized. Assuming fixed wakeup and sleep transition times and energy dissipation values, the total energy consumption of the SP under these two execution sequences is depicted in Fig. 5. It is seen that application scheduling can maximize the SP sleep time.

We must convert the performance constraint for individual applications to those for the individual SPs. The total execution time of an application is the sum of the CPU time, the memory stall time, and the I/O device access time. The throughput of a computer system may then be defined as the ratio of the completed computational workload to the total execution time of the application. Although in a multi-programming system, the calculation of stall time due to I/O devices can be very complicated, it is straight-forward to bound the total I/O stall time by constraining the average delay experienced by each I/O operation. This is because the total I/O stall time is never more than the total I/O operation delay.[1] Based on this observation, we impose constraints on the average service delay of every request sent to each SP to capture the performance constraint on each application.

It is also important to allocate a fair share of the CPU time to each application. In a Linux system, the GPM-based application scheduling algorithm can be implemented using multiple run queues, each associated with a different application type. Based on the decision made by the power manager, at the context switch time, some run queue will be selected and the scheduler will pick one task from this queue to run on the CPU. Inside a run queue, the original priority based scheduling algorithm of Linux kernel is used for task selection. Thus, it is clearly

[1] This is because the I/O operation delay of a request is the waiting time plus the service operation time. The I/O stall time refers to the delay that is encountered during application execution due to IO operations. If there is only one running thread that is stalled after generating each I/O request, the I/O stall time will be equal to the I/O operation delay. However, in a multi-threaded parallel execution environment, the total I/O stall time for the entire batch of executing programs has to be considered. Therefore, the total I/O stall time tends to be less than the total I/O operation delay because some portion of the I/O stall time may be effectively utilized by running other ready applications.

seen that the GPM does not intervene in the scheduling of applications that have the same workload characteristics. The existing fair scheduling schemes [17] such as the FCFS or round robin can be used for these applications. For applications that exhibit different workload characteristics, we must impose a *fairness constraint* as follows. Let $f_r^{a_r}$ denote the frequency that APPL state $r$ is entered and action $a_r$ is chosen in that state, $r \in S_{APPL}$ and $a_r \in A_{APPL}$. Let $\tau_r^{a_r}$ denote the expected duration of time that APPL will stay in state $r$ when action $a_r$ is chosen. Let *flag*(r) denote the flag value component of state $r$. A fairness constraint states that application type $i$ cannot, on average, occupy more than $c_i$ percentage of the CPU time. This can be written as

$$\sum_{r:flag(r)=i} f_r^{a_{r,i}} \tau_r^{a_{r,i}} \le c_i \times 100\%, \quad \text{where } a_{r,i} = run\_Appl_i \qquad (4\text{-}1)$$

where $f_r^{a_{r,i}} \tau_r^{a_{r,i}}$ is the probability that APPL stays in state $r$ and chooses action $a_{r,i}$. One way to determine the value of $c_i$ is to make it proportional to the computation workload of application type $i$. The calculation of $f_r^{a_r}$ and $\tau_r^{a_r}$ actually involves variables and states of other component models in the system, and therefore, it is not convenient to present here. The actual form of this constraint will be given in the section on policy optimization.

## B. *Model of the Service Flow Control*

As illustrated in Fig. 2, the SFC is modeled as a stationary, CTMDP with a state set $S_{SFC}$={*Block*, *Xfer*, *GenF*}and an action set $A_{SFC}$={*Goto_Block*, *Goto_Xfer*, *Goto_GenF*}. The detailed states and transitions of the SFC are explained as follows:

**GenF**: In this state, the SFC generates a fake service request (FSR). An FSR is treated in the same way as a regular SR by the SP, but requires no service from the SP. FSRs are used to wake up the SP when the GPM decides it is the right time to do so. The purpose of FSR is mainly to improve the response time of SP and prevent it from entering a wrong (deep sleep or off) state when the GPM expects a lot of activity in the near future. Delay and energy consumption associated with the transition from Xfer to GenF accounts for the overhead of generating an FSR. The action Goto_Xfer takes place autonomously when the SFC is in GenF.

**Block**: In this state, the SFC blocks all incoming SRs from entering the CQ of the SP. This state may be entered from state Xfer only when all generated SRs have been serviced by the SP. Therefore, when the SFC remains in the Block state, the SSP sees that there are no pending SRs. The purpose of blocking SRs is to reduce the wake-up times of the SP and extend the SP sleep time.

**Xfer**: In this state, the SFC continuously moves SRs from the SQ to the CQ, and therefore, the SP will wake up to provide the requested services. As noted earlier, the CQ is not included in the system-level DPM model, so the function of SFC in the Xfer state is different from its real function, which is described as follows. In this model, when the SFC is in the Xfer state, the SSP knows the status of SQ and FQ and acts the same way that the SP does when the real SRs arrive in the CQ. The time and energy consumption associated with the transition from state Block to Xfer accounts for the overhead of moving about the SRs. The action Goto_Block effects autonomously when and only when the SFC is in Xfer state and SQ and FQ are both empty.

All other state transitions, which have not been mentioned above, take effect immediately and consume no energy.

## C. *Model of the Simulated Service Provider*

The SSP is a CTMDP model that simulates the behavior of the SP under the control of the LPM. Since in the proposed hierarchical DPM architecture, the GPM cannot directly control the state-transition of the SP, the SSP is modeled as an independent automaton. If the LPM employs a CTMDP-based PM policy, then the modeling of SSP will be easy i.e., the CTMDP model of SP with the LPM policy can be used directly, except that the service requests waiting in the SQ and FQ must be considered together when the SSP is making a decision. However, if the LPM employs another PM algorithm, a question will arise as to how accurately a CTMDP SSP model can simulate the behavior of the power-managed SP.

Let's consider an SP with fixed timeout policy, for example, a typical hard disk drive, which has two power states: active at 2.1W and low-power idle at 0.65W. The transition powers and times between the two states are 1.4W and 0.4s. The LPM adopts a two-competitive timeout policy, where the timeout value is set to 0.8s.

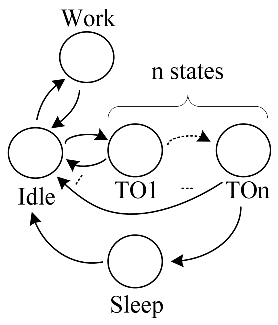

**Fig.6: CTMDP SSP model of HDD with fixed Timeout policy.**

The CTMDP model of the corresponding SSP is depicted in Fig. 6.

**Sleep**: A low-power state. The SSP goes to Idle state when the SFC is in Xfer or GenF state, and the SQ (or FQ) is non-empty.

**Work**: A functional state, where the SSP provides service to the SR that is waiting in the SQ or FQ.

**Idle**: A non-functional state. If the SFC is in either Xfer or GenF states and the SQ (or FQ) is non-empty, the SSP goes to the Work state; otherwise, it goes to TO1 state.

**TO$_i$**: $i$=1,2,…, $n$: One of $n$ full-power but non-functional time-out states. These states are used to simulate the timeout policy. When the SFC is in Xfer or GenF state and the SQ (or FQ) is non-empty, the SSP goes back to the Idle state; otherwise, the SSP goes to TO$_{i+1}$ state or Sleep state if the SSP is in the TO$_n$ state. Since the time for the SSP transferring from Idle to TO$_n$ state is a random variable, while in the timeout policy, the timeout value is fixed, multiple TO states are used to improve the simulation accuracy.

The reason for using multiple $TO_i$ states (instead of just one) is explained as follows. Assume a chain with $n$ TO states is used to approximate a timeout policy whose timeout value is set to $t$. Let $\tau$ denote the time for the SSP transferring from Idle to TO$_n$ state. Let $\tau_0$ and $\tau_1,\ldots,\tau_{n-1}$ respectively denote the time periods that the SSP stays in Idle and TO$_1$,…, TO$_{n-1}$ states when there are no incoming SRs. As required by the CTMDP model, $\tau_0$ and $\tau_1,\ldots,\tau_{n-1}$ are independent random variables, each following an exponential distributions with mean $1/\lambda$ and variance $1/\lambda^2$. To make the expected value of $\tau$ equal to the desired timeout value $t$, it is required that $E(\tau)=n/\lambda=t$, where $\tau=\sum_{i=0}^{n-1}\tau_i$. Thus, variance of $\tau$ is $D(\tau)=\sum_{i=0}^{n-1}D(\tau_i)=n/\lambda^2=t^2/n$. From this equation, we can see that for a given $t$, as $n$ increases, $D(\tau)$ is reduced. In other words, the accuracy of the CTMDP model of a fixed timeout policy increases.

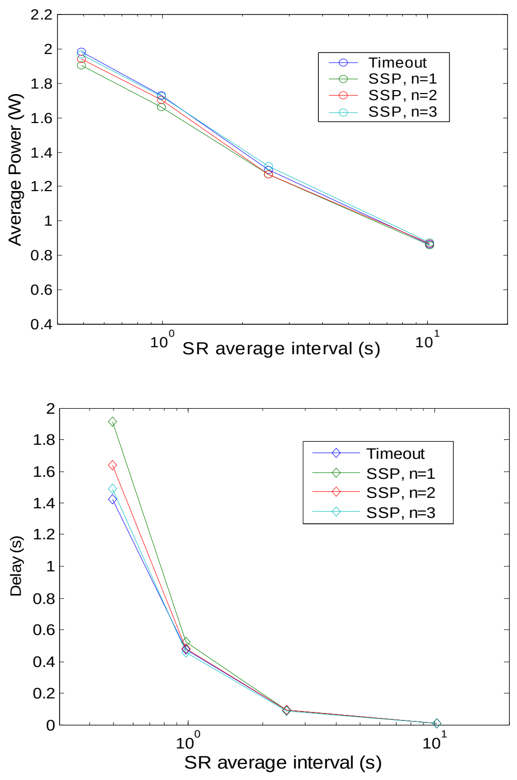

**Fig. 7. Comparison between the CTMDP model of the SSP and the fixed timeout policy for the hard disk.**

We performed a simulation study to evaluate how the approximation accuracy is related to the number of TO states in the SSP model in terms of energy and service delay for the abovementioned hard disk example. Results are presented in Fig. 7. The average power and delay of the SP under a fixed timeout policy is compared with three SSPs, which respectively use one, two and three TO states to simulate the same timeout policy. It is demonstrated that, with three TO states, behavior of the SSP becomes indistinguishable from that of the hard disk with a fixed timeout policy.

## D. *Modeling Dependencies between SPs*

There are different types of dependency between SPs. The first type is mutual exclusion. Mutual exclusion arises for example, when two SPs contend for the same non-sharable resource, e.g., a low speed I/O bus. Consequently, at any time, only one SP can be in its working state. When constructing the CTMDP model of the system, one can account for this type of hard dependency constraint by marking any system state that violates the mutual exclusion as invalid and by forbidding all state-action pairs that cause the system to transit to an invalid state. The second type is shared resource constraint, where two SPs indirectly influence one another's behavior because of their utilization of a shared resource. For example, SPs may want to buffer their SRs in a shared buffering area of finite size. So when the number of SRs for one SP goes up, the probability that SRs for the other SP will be blocked increases. In this case, the first SP may have to work harder to ensure that it is not over-utilizing the shared buffer area. This type of soft dependency constraint is handled by adding appropriate constraints to the system-level power optimization problem formulation.

## V. Policy Optimization

### A. *Mathematical Formulation*

Let *I* denote the number of SPs in the power-managed system. Let *x* represent the global state of this system, which is a vector whose elements are the states of the APPL, SQ*i*, SFC*i* and SSP*i* models, with *i*=1,2,…*I*. Let $a_x$ denote an action enabled in state *x*, which is a tuple composed of the actions of the APPL and $\text{SFC}_i$ models. The constrained energy optimization problem is formulated as a linear program as follows:

$$\text{Minimize}_{\{f_x^{a_x}\}}\left(\sum_x\sum_{a_x} f_x^{a_x}\gamma_x^{a_x}\right) \qquad (5\text{-}1)$$

where $f_x^{a_x}$ is the frequency that global state *x* is entered in and action $a_x$ is chosen in that state. $\gamma_x^{a_x}$ is the expected cost, which represents the expected energy consumed when the system is in state *x* and action $a_x$ is chosen, is calculated as:

$$\gamma_x^{a_x} = \tau_x^{a_x}\, pow(x,a_x) + \sum_{x'\neq x} p_{x,x'}^{a_x}\, ene(x,x') \qquad (5\text{-}2)$$

where $\tau_x^{a_x} = 1\Big/\sum_{x'\neq x}\sigma_{x,x'}^{a_x}$ denotes the expected duration of time that the system will stay in state *x* when action $a_x$ is chosen and $\sigma_{x,x'}^{a_x}$ is the rate of the transition from state *x* to state *x'* when action $a_x$ is chosen. In addition, $p_{x,x'}^{a_x} = \sigma_{x,x'}^{a_x}\Big/\sum_{x''\neq x}\sigma_{x,x''}^{a_x}$ denotes the probability that the system will next come to state *x'* if it is in state *x* and action $a_x$ is chosen. This linear program is solved for variables $f_x^{a_x}$ while satisfying the constraints given below.

$$\sum_{a_x} f_x^{a_x} = \sum_{x'\neq x}\sum_{a'_x} f_{x'}^{a'_x} p_{x',x}^{a'_x} \qquad \forall x\in X \qquad (5\text{-}3)$$

$$\sum_x\sum_{a_x} f_x^{a_x}\tau_x^{a_x} = 1 \qquad (5\text{-}4)$$

$$f_x^{a_x} \geq 0 \qquad (5\text{-}5)$$

$$\sum_x\sum_{a_x} f_x^{a_x}\tau_x^{a_x}(q_{i,x} - D_i\lambda_{i,x}) \leq 0, \quad i=1,2,\dots,I \qquad (5\text{-}6)$$

$$\sum_{x:\, flag(r_x)=i}\ \sum_{a_{r,j}\in a_x} f_x^{a_x}\tau_x^{a_x} \leq c_j \times 100\%, \quad j=1,2,\dots,J \qquad (5\text{-}7)$$

where $r_x$ denotes state of APPL in global state *x* and $a_{r,j} = run\_Appl_j$.

$$\sum_x\sum_{a_x} f_x^{a_x}\tau_x^{a_x}\delta(q_{i,x},Q_i) \leq P_{i,b} \quad i=1,2,\dots,I \qquad (5\text{-}8a)$$

or $$\sum_x\sum_{a_x} f_x^{a_x}\tau_x^{a_x}\delta(\sum_i q_{i,x},Q) \leq P_b \quad \text{with a shared Q} \qquad (5\text{-}8b)$$

where $\delta(x,y) = \begin{cases}1, & \text{if } x=y;\\ 0, & \text{otherwise.}\end{cases}$.

Equations (5-3) through (5-5) capture properties of a CTMDP. Inequalities (5-6), based on the Little's theorem [18], impose constraints on the expected task delay of $\text{SP}_i$, where $q_{i,x}$ represents the number of waiting tasks in the queue $\text{SQ}_i$ when the system is in state *x*, $D_i$ is the expected service delay experienced by $\text{SR}_i$, and $\lambda_{i,x}$ is the generation rate of the $SR_i$ at system state $x$. Inequalities (5-7) are the same as (4-1) and state that on average application type $j$ should not use more than $c_j$ percent of the CPU time. $J$ is the number of application types in APPL. Constraint (5-8a) and (5-8b) ensure that the probability that the SQ becomes full is less than a preset threshold. Constraint (5-8a) is imposed when each type of SR utilizes its own non-sharable SQ, while constraint (5-8b) is applied when a shared SQ is used for all types of SRs. This linear program is solved by using a standard solver, i.e., MOSEK [19].

### *B. Optimal Timeout Policy for Local Power Manager*

The DPM optimization discussed up to now assumes that the timeout policy used for local power manager has been given and is unable to change. However, in many real cases, the embedded power management solutions provide mechanisms for user to tune the local policy parameters. For example, Windows power manager provides multiple optional schemes and allow users to change the length of idle duration that triggers to enter lower power mode. Also, some latest WLAN cards can be configured to wake up every so many multiples of the beacon intervals and the length of a beacon interval is negotiable. In terms of this observation, we define an optimization problem that simultaneously optimizes the system-level power management policy and the timeout policy of the local power manager.

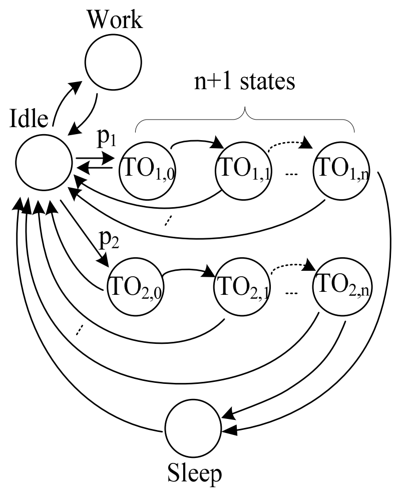

**Fig. 8: CTMDP SSP model of HDD for timeout optimization.**

The optimization of timeout values can not be directly incorporated into CTMDP-based DPM optimization framework. A perturbation analysis based timeout optimization technique was proposed in [20]. Here, we are proposing an indirect approach to obtain a near-optimal solution for timeout values. For this purpose, the CTMDP SSP model that was presented in Fig. 6 has been modified as shown in Fig. 8. In this model, there are two TO state chains starting from a low-power non-functional state, e.g. Idle state. The two chains correspond to different timeout values, denoted as $t_1$ and $t_2$, respectively. Apart from the SSP model in Fig. 6, a $TO_{*0}$ state is added to each TO chain representing the start of an idle period. Once the model enters the idle state, if no task is queuing, it immediately transfers to $TO_{1,0}$ state with probability $p_1$ or $TO_{2,0}$ state with probability $p_2=1-p_1$ and takes no time. This models the behavior that during each idle period, timeout value $t_1$ is taken with probability $p_1$, and timeout $t_2$ with probability $p_2$. Here, $p_1$ and $p_2$ are controlling parameters to be optimized with the system-level DPM policy. Once they are solved, the optimal timeout value $t_{opt}$ can be approximated by

$$t_{opt} \approx p_1 t_1 + p_2 t_2 \,. \tag{5-9}$$

To combine the optimization of local timeouts with system-level DPM policy and use the linear programming approach, we need define an action set for the SSP model, as $A_{SSP}$={$Goto\_TO_{1,0}$, $Goto\_TO_{2,0}$}. Now, the system action $a_x$ should be a tuple composed of the actions of the APPL, SFC, and SSP models. Thus, after updating the linear program (5-1) to (5-8) accordingly and solving the optimization problem, we can obtain the values of $p_1$ and $p_2$ as

$$p_1 = \sum_{\substack{q=0,s=Idle,\\ a_x=Goto\_TO_{1,0}}} f_x^{a_x} \Bigg/ \sum_{q=0,s=Idle,} f_x^{a_x}, \quad p_2 = 1 - p_1, \qquad (5\text{-}10)$$

where $q$ and $s$ denote the state of the SQ and SSP when the system is in state $x$, respectively. Equation (5-9) is next employed to determine the optimal timeout values of the local policy. It is worth noting that to achieve a good approximation of the optimal value, the $t_1$ and $t_2$ values should be selected carefully to ensure that $t_{opt}$ is between $t_1$ and $t_2$. This can be achieved by using an iterative approach. In case that either $p_1$ or $p_2$ is close to 1, we choose a new set of timeout values and redo the optimization. Assuming $p_1$=1 and $t_1$<$t_2$, the new timeouts will be

$$t_1' = \max(0, t_1 - t_2'), \quad t_2' = (t_1 + t_2)/2. \qquad (5\text{-}11)$$

Further, we can simply use more than two TO state chains for a low-power physical state to improve the accuracy.

Notice that the SSP model presented in Fig. 8 is for a device that has only one low power non-functional state, i.e. a “Sleep” state as marked in the figure. The two timeout values $t_1$ and $t_2$, which are approximated by the two TO state branches in the SSP model in Fig. 8, are predetermined values used to calculate the optimal timeout value $t_{opt}$ that will be employed by the local power manager to control the state transition from idle to sleep state. For a device with two low power states, i.e. a shallow sleep (drowsy) and a deep sleep state, this simple model may be extended in a straight-forward manner by adding a pair of TO states between every two neighboring system states in order to capture the timeout state transition between the corresponding system states.

## VI. EXPERIMENTAL RESULTS

For this experiment, we recorded a real trace of device requests generated by four concurrently running applications on a Linux PC. The applications were of two types. Three of them were file manipulation programs, which read some data file, edit it and write back to the disk. The fourth application was a program which periodically reads data from another machine through a WLAN card, searches for relevant information, and saves this information onto the disk. The request generation pattern of the first type of application was modeled with a Poisson process with an average rate of 0.208 requests per second. The request generation statistics of the second program type can be best characterized by a two-state CTMDP model. The state transition rate and generation rates of SR to hard disk $\lambda_{hd}$ and to WLAN card $\lambda_{wlan}$ are

$$\begin{bmatrix} 0 & 0.0415 \\ 0.0063 & 0 \end{bmatrix} \left(s^{-1}\right), \quad \begin{matrix} \lambda_{hd} = [0.0826, 0.0187] \\ \lambda_{wlan} = [0.1124, 0.1124] \end{matrix} \left(s^{-1}\right).$$

The CPU usage ratio for these two groups of applications (i.e., two application types) is 53:47. For our experiments, we used the hard disk drive Hitachi Travelstar 7K60 and Orinoco WLAN card as service providers. Power dissipation and start-up energy and latency of the disk drive and the WLAN card are reported in Table 1.

TABLE 1
ENERGY/TRANSITION DATA OF HARD DISK DRIVER AND WLAN CARD

| | State | Power (w) | Start-up energy (J) | Wake-up time (s) |
|---|---|---|---|---|
| Hitachi 7K60 | Active | 2.5 | -- | -- |
| | Performance idle | 2.0 | 0 | 0 |
| | Low power idle | 0.85 | 1.86 | 0.4 |
| | Stand-by | 0.25 | 10.5 | 2 |
| Orinoco WLAN | Transfer | 1.4 | -- | -- |
| | Receive | 0.9 | -- | -- |
| | Sleep | 0.05 | 0.15 | 0.12 |

For the first set of simulations, we only consider the hard disk driver. The average service time for a disk request is 67ms. In this case, with the help of the operating system, fake service request (FSR) can be designed as a disk read operation that accesses the latest data read from the hard disk. Since this data must have been stored in the data cache of the hard disk, it does not have to be read out from the disk, so the service time of an FSR is only the sum of disk controller's overhead and the data transfer time, which is about 3ms.

We used the lower envelope algorithm [21], which is a 2-competitive policy extended for a device with multiple low-power states, as the timeout policy for the LPM. The LPM policy has 2 timeout values, each corresponding to one low power state. They are 1.7s and 14.4s. Under this policy (named TO1), the SP starts in the highest power state ("Active"="Performance idle".) If there are no new requests, after 1.7s elapses, it enters "Low power idle" state. If no requests arrive, after 14.4s, it enters into its "Stand-by" state. We also experimented with a different set of timeout values, i.e., 0.34s and 14.4s. This version is denoted by TO2. Results are presented in Table 2.

TABLE 2
HIERARCHAL PM SIMULATION RESULTS FOR SINGLE SP

| CPU usage | LPM policy | Perf. Cons. | 1PM-TO(W) | 1PM-CT MDP (W) | HPM (W) | HPM-S (W) |
|---|---|---|---|---|---|---|
| 0.53: 0.47 | TO1 | 0.0765 | 1.2728 | 1.0467 | 1.2591 | 0.9505 |
| | | 0.5 | 1.2728 | 0.9309 | 1.0443 | 0.788 |
| | TO2 | 0.0882 | 1.1582 | 1.0414 | 1.1436 | 0.8651 |
| | | 0.5 | 1.1582 | 0.9309 | 1.0106 | 0.7274 |
| 0.7: 0.3 | TO1 | 0.078 | 1.3805 | 1.1152 | 1.342 | 0.9951 |
| | | 0.5 | 1.3805 | 0.9956 | 1.1047 | 0.8302 |
| | TO2 | 0.0903 | 1.2559 | 1.1107 | 1.2032 | 1.0594 |
| | | 0.5 | 1.2559 | 0.9956 | 1.0966 | 0.8734 |
| 0.3: 0.7 | TO1 | 0.0685 | 1.19 | 0.9647 | 1.1058 | 0.957 |
| | | 0.5 | 1.19 | 0.7922 | 0.9276 | 0.788 |
| | TO2 | 0.076 | 1.0162 | 0.9451 | 1.012 | 0.7373 |
| | | 0.5 | 1.0162 | 0.7922 | 0.8422 | 0.6015 |

In the above table, the first column gives the CPU usage ratio between the two types of applications. The type of the built-in LPM policy is reported in the second column. For each LPM policy, we simulate twice for different performance constraints in terms of the bound on the average number of waiting SRs in the SQ. This bound is reported in the third column. In each case, the smaller bound corresponds to the actual SR delay in the timeout policy simulation. The second one is a looser constraint given for the purpose of examining the ability of our proposed hierarchical DPM approach to trade off latency for lower energy consumption.

Four policies are compared in this table, they are one-level timeout policy (1PM-TO), one-level CTMDP policy (1PM-CTMDP), Hierarchical PM (HPM) and HPM with application scheduling (HPM-S). For the stochastic policies, the SR generation statistics is assumed to be known. The average power consumptions of the SP under different policies are reported in the last four columns of the table. Comparing HPM with local power management policy 1PM-TO, it can be seen that HPM improves the energy efficiency of LPM controlled service provider, especially when there is a large positive slack in which case up to 22% energy saving is achieved. This saving is made possible because of the system-level service flow control policy incorporated in HPM, which monitors the state of local power manager and subsequently adjusts the rate of SRs sent to the SP in order to make the SP run more power efficiently and increase the chance that it stays in its lower power states. HPM-S even outperforms the optimal component-level CTMDP policy by as much as 24% in terms of energy consumption saving.

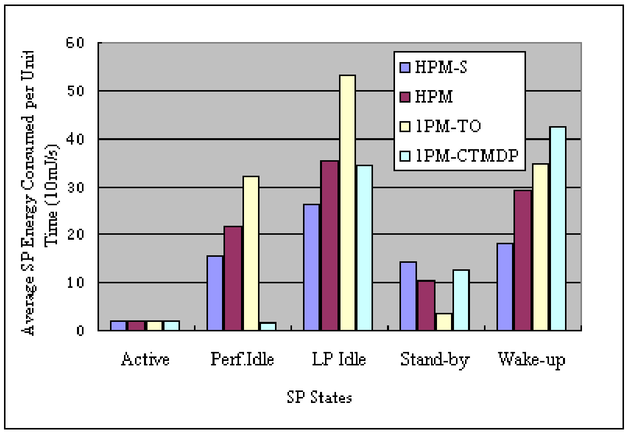

**Fig. 9: Break-down of the power consumption of the service provider.**

For better understanding of how the energy saving is achieved by HPM policies, the break-down of the total power consumptions of the SP under different policies are presented in Fig. 9, where TO1 is employed as the local policy with the CPU usage ratio 0.53:0.47 and the performance constraint set to 0.5. As compared to the timeout policy, HPM and HPM-S significantly reduced the SP energy consumed at high-power idle states and dissipated for wake-ups by a total amount of 33.9mJ and 60.3mJ per second respective, with a small increase in average standby power by 6.9mW and 10.8mW due to SP staying longer in standby mode. From this figure, it is demonstrated that HPM approaches allow the SP to spend more time at the lowest power state while reducing wakeup overhead simultaneously. When compared to 1PM-CTMDP, HPM and HPM-S reduce the wake-up energy dissipation at the cost of extra energy consumed in high-power idle states. This difference lies in the fact that 1PM-CTMDP makes a decision to transit to a low power state immediately as soon as it becomes idle whereas HPM and HPM-S must wait until a local timeout counter expires.

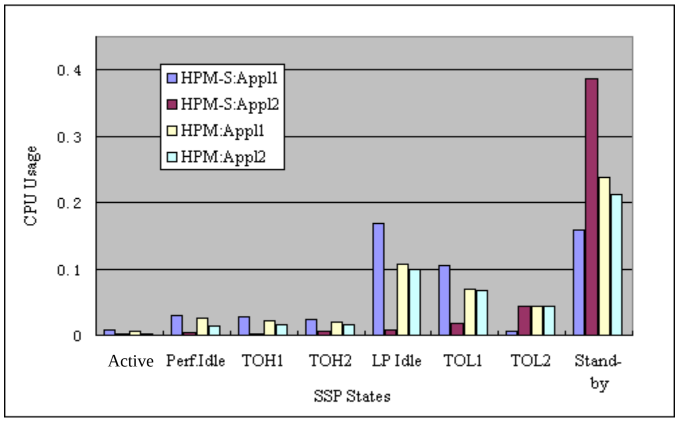

**Fig. 10: Break-down of CPU usage of applications under HPM policies.**

The application-level scheduling incorporated into HPM selects applications to run based on the global system state, i.e., states of the SP and the SQ, and dynamically adjusts the SR generation rate to help reduce the SP state-transition times and increase the duration of time that the SP stays in low power states, while meeting the given timing and fairness constraints. To emphasize on the effect of application scheduling on power management, the CPU usage of each application type is divided into bins corresponding to the SSP states and compared in Fig. 10 between HPM and HPM-S policies, where the simulation setup is the same as that used to generate Fig. 9. The labels on the *x*-axis, TOH*n* and TOL*n*, *n*=1,2, respectively represent the timeout states while the SP is in Hitachi performance-idle and low-power-idle state. Please note that in Fig. 9, energies consumed in the TOH and TOL states are added to those consumed in Perf. Idle and LP Idle states, respectively.

In this figure, each bar represents the CPU usage of an application type with respect to an SSP state, which equals to the time when the application is running while the SSP is in the associated state divided by the overall running time of all application types. As shown in this figure, without application scheduling, the two application types Appl1 and Appl2 have very close CPU usages on all SSP states.  However, under HPM-S policy, at SSP highest power states the CPU usages of Appl1 are much higher than Appl2, while the reverse exists at the lowest-power state Standby. Appl1 has a higher SR generation rate than Appl2. Executing Appl1 than Appl2 at high power state will make more likely that a new SR is generated while the SP is still in a high power state. In this case, the SP will be easier to transit back to Work state with less energy dissipated for state-transition and faster response time. On the other side, performing Appl2 at a state closer to the lowest power state is likely to increase the interval before the next SR, and thus create more idle duration for the SP to stay in the lowest power state and reduce the energy consumption.

TABLE 3
HIERARCHAL PM SIMULATION RESULTS FOR SINGLE SP WITH OPTIMAL LOCAL TIMEOUT POLICY

| CPU usage ratio | Perf. Cons. | Opt. TOH(s) | Opt. TOL(s) | HPM (W) | Opt. TOH(s) | Opt. TOL(s) | HPM-S (W) |
|---|---|---|---|---|---|---|---|
| 0.53: | 0.088 | 0.035 | 50.0 | 1.057 | 0.024 | 22.3 | 0.868 |
| 0.47 | 0.5 | 0.019 | 15.7 | 0.887 | 0.020 | 13.1 | 0.697 |
| 0.7: | 0.090 | 0.028 | 63.1 | 1.125 | 0.027 | 43.4 | 1.000 |
| 0.3 | 0.5 | 0.023 | 30.2 | 0.996 | 0.022 | 21.8 | 0.838 |
| 0.3: | 0.076 | 0.021 | 41.0 | 0.978 | 0.017 | 10.6 | 0.709 |
| 0.7 | 0.5 | 0.016 | 10.0 | 0.731 | 0.015 | 10.0 | 0.563 |

In the second set of simulations, we still considered a single service provider but exploited the technique presented in section V.B to determine the optimal timeout values for the local power manager. The results are presented in Table 3. In this table, the obtained optimal timeout values are listed before the corresponding HPM policy. As compared to the results in Table 2, it is observed that for HPM-S policy, using optimal local timeouts does not incur much energy saving. The main reason is that the incorporated application scheduling technique is able to counteract the impairment introduced by an imperfect local timeout policy. However, HPM policy did benefit from an optimal local timeout and improve the energy saving by 8.7% on average.

TABLE 4
HIERARCHAL PM SIMULATION RESULTS FOR TWO SPS

| | Perf. Cons. for different SPs | | 1PM - TO2 (W) | 1PM - CTMDP (W) | HPM (W) | HPM-S (W) |
|---|---|---|---|---|---|---|
| Sim1 | HD | 0.09 | 1.157 | 1.045 | 1.142 | 0.881 |
| | WLAN | 0.05 | 0.384 | 0.343 | 0.378 | 0.310 |
| Sim2 | HD | 0.2 | 1.157 | 1.01 | 1.066 | 0.788 |
| | WLAN | 0.2 | 0.384 | 0.322 | 0.331 | 0.282 |

In the third set of simulations, we considered two service providers: a hard disk and a WLAN card. The average service time for a wireless request is 830ms. In this simulation, policy TO2 is used for the LPM of the hard disk driver and a 2-competitive policy with a timeout value of 200ms is used for the WLAN card. The WLAN card also wakes up every second to listen for traffic identification message. We used the SR trace with a CPU usage ratio 53:47 in this simulation. The results of the power consumption of each component are presented in Table 4. The experiment results demonstrate that the HPM-S algorithm can jointly schedule applications for different SPs to achieve minimal total system energy consumption.

## VII. CONCLUSION

This article presented an HPM architecture which aims to facilitate power-awareness in an EMC system with multiple components. Given a performance constraint, this architecture improves both component-level and system-wide power savings by using information about service request rates by tuning the PM policies of components. The technique to obtain an optimal timeout for LPM is also presented. Experimental results demonstrate that the system-level PM approach can result in significant extra energy savings.

An interesting direction for future work is to extend hierarchical power management approach to handle non-stationary service request generation. One possible solution is to construct offline a policy tree where each leaf node represents an optimal policy for a given set of system parameters, e.g. delay constraint, request generation rates and CPU share of different application types. At run time, system parameters will be dynamically detected and used as an index in the policy tree. The policy table that matches the current settings will be exploited.

## References

[1] 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, No. 3, pp. 42–55, Mar. 1996.

[2] C-H. Hwang and A. Wu, "A predictive system shutdown method for energy saving of event-driven computation," *Proc. Int. Conf. Computer-Aided Design*, pp. 28–32, Nov. 1997.

[3] L. Benini, G. Paleologo, A. Bogliolo, and G. De Micheli, "Policy optimization for dynamic power management," *IEEE Trans. Computer-Aided Design*, Vol. 18, No. 1, pp. 813–33, Jun. 1999.

[4] Q. Qiu, Q Wu and M. Pedram, "Stochastic modeling of a power-managed system-construction and optimization," *IEEE Trans. Computer-Aided Design,* Vol. 20, No. 10, pp. 1200-1217, Oct. 2001.

[5] U. N. Bhat, *Elements of Applied Stochastic Processes*. New York: Wiley, 1984.

[6] Y-H. Lu, L. Benini and G. De Micheli, "Power-aware operating systems for interactive systems," *IEEE Trans. VLSI System*, Vol.10, no. 4, pp. 119-134, Apr. 2002.

[7] Storage Systems Division, IBM Corp., APM for Mobile Hard Disks, www.almaden.ibm.com/almaden/mobile_hard_drives.html, 1999.

[8] T. Simunic, L. Benini, P. Glynn, G. De Micheli, "Event-driven power management," *IEEE Trans. Computer-Aided Design*, Vol. 20, No. 7, pp. 840-857, Jul. 2001.

[9] G. Theocharous, S. Mannor, N. Shah, and et al., "Machine learning for adaptive power management," *Intel Technology Journal*, Vol. 10, Iss. 4, pp. 299-312, Nov. 2006.

[10] T. Simunic, S. Boyd, and P. Glynn, "Managing Power in Networks on Chips", *IEEE Trans. VLSI Systems*, Vol.12, No. 1, pp. 96-107, Jan. 2004.

[11] Z. Ren, B.H. Krogh, and R. Marculescu, "Hierarchical adaptive dynamic power management," *IEEE Trans. Computer*s, Vol. 54, No. 4, pp. 409–420, April 2005.

[12] H. Jun, M. Ammar, M. Corner and et al., "Hierarchical Power Management in Disruption Tolerant Networks with Traffic-Aware Optimization," *Proc. SIGCOMM Workshop on CHANTS*, pp. 245-252, Sept. 2006.

[13] J. Sorber, N. Banerjee, M. Corner and et al., "*Turducken: Hierarchical Power Management for Mobile Devices*," *Proc. MobiSys*, pp. 261-274, Seattle, Jun. 2005.

[14] U. N. Bhat, *Elements of Applied Stochastic Processes*, New York: Wiley, 1984.

[15] K. Choi, K. Soma, and M. Pedram, "Fine-grained DVFS for precise energy and performance trade-off based on the ratio of off-chip access to on-chip computation times," *Proc. Design and Test in Europe*, pp. 4–9., Feb. 2004.

[16] V. Delaluz, A. Sivasubramaniam, M. andemir, N. Vijaykrishnan and M. J. Irwin, "Scheduler-based DRAM energy management," *Proc. of Design Automation Conf.*, pp. 697–702, Jun. 2002.

[17] A. Silberschatz, P.B. Galvin, G. Gagne, *Operating System Concepts*, John Wiley & Sons, 2004.

[18] E. A. Feinberg, A. Shwartz, *Handbook of Markov decision processes: methods and applications*, Kluwer Academic, 2002.

[19] E. D. Andersen and K. D. Andersen, "The MOSEK interior point optimizer for linear programming: an implementation of the homogeneous algorithm", in *High Performance Optimization*, pp. 197-232. Kluwer Academic, 2000.

[20] 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. Design and Test in Europe*, pp. 1128-1133, Mar. 2006.

[21] S. Irani and S. Shukla and R. Gupta. "Competitive analysis of dynamic power management strategies for systems with multiple power saving states," *Proc. Design and Test in Europe*, pp.117-23, Feb. 2002.