Hierarchical Dynamic Power Management with Application Scheduling

Abstract *- This paper proposes a hierarchical power management architecture which aims to facilitate power-awareness in an Energy-Managed Computer (EMC) system with multiple (potentially self-power-managed) components. Given a performance constraint for the system, this architecture improves both component-level and system-wide power savings by using information about service request rates obtained from the operating system by dynamically guiding/tuning the local power management policies of the components. The proposed architecture divides power management function into two layers: system-level and component-level. The system-level power management will be formulated as a service request flow regulation problem. In addition, application scheduling will be integrated with system-level power management to achieve higher power savings. The overall system model is constructed as a stochastic controllable process based on the theory of continuous-time Markovian decision processes (CTMDP). Experimental results demonstrate that the system-level power management approach can result in significant extra energy savings compared to energy savings that can be achieved by the component-level power management.*

1 Introduction

Dynamic power management (DPM), which refers to a selective shut-off or slow-down components that are idle or underutilized, has proven to be a particularly effective technique for 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.

Previous work on stochastic DPM techniques has focused on developing component level policies, where the power-managed component is assumed to be operating independently of the other components of the system. For example, these techniques cannot account for a mutual-exclusion relationship that will exist between the working states of two components if their operations require the same non-sharable resource. In addition, the previous work has not differentiated between the service request characteristics of different software applications, and therefore, it has effectively ignored the potential benefit of performing application-level scheduling as part of the power management process.

Reference [5] 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 [6]. 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. The problem is then formulated as a mathematical program with the aid of CTMDP models and solved accordingly. Key contributions of this paper may be summarized as follows. (1) A hierarchical DPM architecture is proposed, where the power management function is divided into system and component levels. 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 based on states of the individual components, the number of waiting tasks, and application stochastic characteristics, which makes it very different from [5]. (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.

In the literature, some works related to the hierarchical power management have been reported. Reference [7] 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 paper 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 [8] 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 paper. 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.

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

2 Background

The continuous-time Markovian decision processes (CTMDP) based DPM approach was proposed in [4]. CTMDP-based approach makes policy changes in an asynchronous and event-driven manner and thus surmounts the shortcoming of the earlier work based on discrete-time Markovian decision processes [3], which relied on periodical policy evaluation. We believe CTMDP is more suitable for implementation as part of a real-time operating system environment because of its even-driven nature. A CTMDP model is defined with a discrete state space; a generator matrix, where an entry represents the transition rate from one state to another; an action set; and a reward function. In CTMDP, the generator matrix is a parameterized matrix that depends on the selected action. A complete system may comprise of several components, each modeled by a CTMDP. The state set of the complete system is obtained as the Cartesian product of the state set of each component minus the set of invalid states. The generator matrix of the whole system can be generated from the generator matrices of its components by using the tensor sum and/or product operations. Due to space limitation, the details of CTMDP modeling technique cannot be presented here. Interested readers may refer to [4].

3 A Hierarchical DPM Architecture

In this paper, 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 fair share of CPU time over a long period of time. Our objective is to minimize the energy consumption of the computer system. More precisely, this paper focuses on reducing energy consumption of the I/O devices. Saving processor and memory energy is out of the scope of this paper. Readers interested in these power components can refer to [9] and [10].

The architecture of our proposed hierarchical DPM framework which contains two service providers (SP), i.e. two I/O devices, is presented in Figure 1. This architecture has two levels of power management. In the component level, each SP is controlled by a local power manager (LPM). The LPM performs a conventional power management 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. First, the 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 power management policy and improve the power efficiency of the SPs. Second, the GPM works with the CPU scheduler to select the right applications to run on the CPU in order to reduce the expected system power dissipation. This decision is in turn made based on the current state of the power management system, including the states of the SPs and the number of SRs waiting in the SQ. Third, 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, in order to adjust the statistics of the service request flow that reaches the SP. The SRs that are blocked by the SFC are stored in a service queue (SQ).

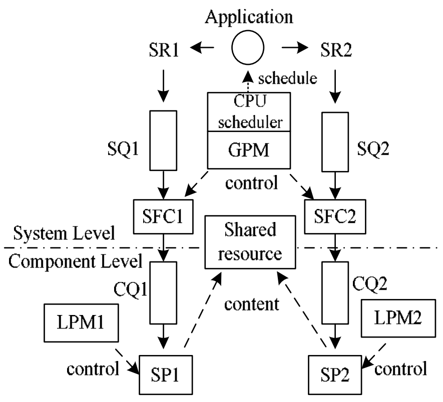

Figure 1. Block diagram of a hierarchical DPM structure.

4 Modeling

We represent the hierarchical DPM structure by a CTMDP model as shown in Figure 2. This model, which is constructed from the point of view of the GPM, is utilized to derive a system-level power management 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. Note that the transition diagram of the SSP shown in Figure 2 is an example used for illustration purposes.

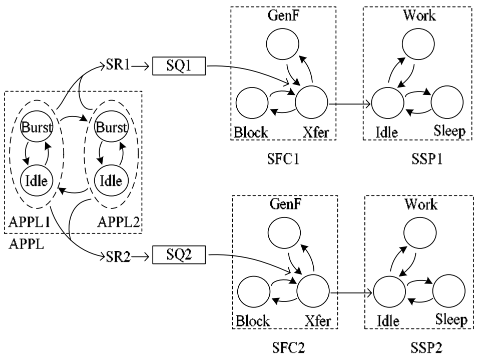

Figure 2. CTMDP model of the hierarchical DPM structure.

4.1 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 [8], 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 shown in Figure 3 , in state $r1_a$, 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 $r1_a$, application type 1 transits to state $r1_b$ with a rate of $\upsilon 1,_{ab}$, which also implies that the average time for application type 1 to stay in state $r1_a$ is $1/\upsilon 1,_{ab}$.

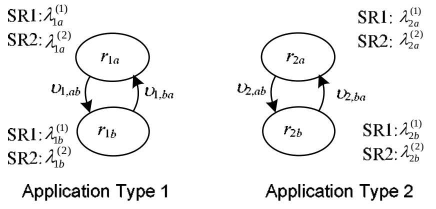
SR1: $\lambda_{1a}^{(1)}$
SR2: $\lambda_{1a}^{(2)}$
r_{1a}
$\upsilon_{1,ab}$
$\upsilon_{1,ba}$
SR1: $\lambda_{1b}^{(1)}$
SR2: $\lambda_{1b}^{(2)}$
r_{1b}
Application Type 1
SR1: $\lambda_{2a}^{(1)}$
SR2: $\lambda_{2a}^{(2)}$
r_{2a}
$\upsilon_{2,ab}$
$\upsilon_{2,ba}$
r_{2b}
SR1: $\lambda_{2b}^{(1)}$
SR2: $\lambda_{2b}^{(2)}$
Application Type 2

Figure 3. CTMDP models of application types 1 and 2.

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 which application is currently running on the CPU. The CTMDP model of example S_{APPL} shown in Figure 4 has eight global states, ($r1_x$,$r2_y$,*flag*) where $r1_x$ denotes the service generation state *x* for application 1 whereas $r2_y$ denotes the service generation state y for application type 2. *flag*=1 means that the first application is running while *flag*=2 means that the second application is running. For example, ($r1_a$,$r2_a$,1) means that application type 1 is running and it is in state $r1_a$. Furthermore, the state of application type 2 was $r2_a$ 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 ($r1_a$,$r2_a$,1) and ($r1_b$,$r2_a$,1) is autonomous. Notice that a transition from ($r1_a$,$r2_a$,1) and ($r1_b$,$r2_b$,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 $r1_x$,$r2_y$ values.

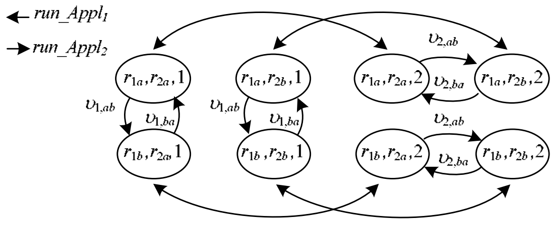

Figure 4. CTMDP model of an application pool.

The action set is A_{APPL} = {*run_Appl*$_i$}, 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). Note that a transition between (r_{1a},r_{2b},1) and (r_{1a},r_{2a},2) in not allowed because it would imply that during context switch from application type 1 to application type 2, the service generate state of application 2 changed, an impossibility in our model.

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 Figure 5. It is seen that application scheduling can maximize the sleep time of the SP.

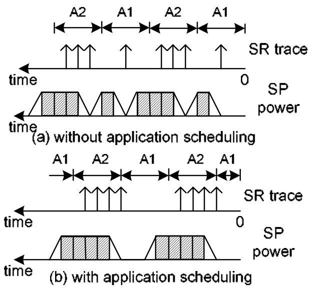

Figure 5. An example of the effectiveness of application scheduling.

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 always less than the total I/O operation delay. Based on this observation, in this paper, we shall impose constraints on the average service delay of every request sent to each SP in order to roughly capture the performance constraint on each application.

It is also important to allocate a fair share of the CPU time to each application. Because the GPM does not intervene in the scheduling of applications that have the same workload characteristics, existing fair scheduling schemes [11] 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.

4.2 Model of the Service Flow Control

As illustrated in Figure 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 actual 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. The 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 state.

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 extends the SP sleep time.

Xfer: In this state, the SFC continuously moves the SRs from the SQ to the CQ and therefore, the SP will wakes up to provide the requested services. As explained before, the CQ is not included in the system-level DPM model, so the function of SFC at Xfer state is slightly different from its real function, which is describe 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.

4.3 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 power management 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 should be considered together when the SSP is making a decision. However, if the LPM employs another power management algorithm, a question will arise as to how accurate a CTMDP SSP model can simulate the behavior of the power-managed SP. Since time-out policy is the most widely used power management policy in commercial electronic systems, we focus our discussion on building the SSP model for a timeout policy-based SP.

Let's consider an SP with fixed timeout policy. For example, assume a hard disk drive as the SP. It 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, respectively. The LPM adopts a two-competitive timeout policy, where the timeout value is set to 0.8s. The CTMDP model of the corresponding SSP is depicted in Figure 6.

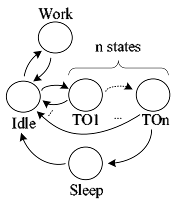

Figure 6. CTMDP SSP model of HDD with fixed Timeout policy.

The detailed states and transitions of the SSP are explained next.

Sleep: A low-power deep-sleep 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 currently 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 τ denote the time for the SSP transferring from Idle to TO_n state. Let τ_0 and $\tau_1,\ldots,\tau_{n-1}$ respectively denote the time periods that the SSP stays in Idle and TO1,…, TO_n-1 states when there are no incoming SRs. As required by the CTMDP model, τ_0 and $\tau_1,\ldots,\tau_{n-1}$ are independent random variables, each following an exponential distributions with mean $1/\lambda$ and variance $1/\lambda^2$. It is obvious that $\tau = \sum_{i=0}^{n-1} \tau_i$. To make the expected value of τ equal to the desired timeout value *t*, it is required that $E(\tau) = n/\lambda = t$. Thus, variance of τ 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(τ) is reduced. In other words, the accuracy of the CTMDP model of a fixed timeout policy increases.

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. The results of the hard disk under timeout policy and its SSP models with respect to different SR average intervals are reported in Figure 7. It is seen that with three TO states, the behavior of the SSP becomes indistinguishable from that of the hard disk with a fixed timeout policy.

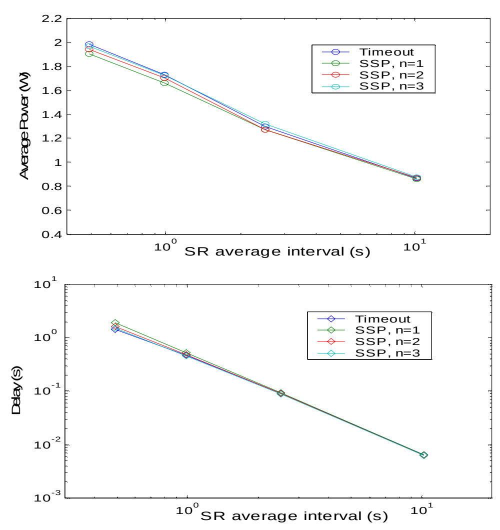

Figure 7. Comparison between the CTMDP model of the SSP and the fixed timeout policy for the hard disk.

4.4 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, both SPs may want to buffer their SRs in a shared buffering area with a finite size. Consequently, when the number of SRs for one SP goes up, the probability that the 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.

5 Policy Optimization

As discussed in section 4.1, the system-level optimal policy is solved under constraints on the expected SR service time, the SR blocking rate, and the CPU utilization time constraint. We formulate the policy optimization problem as a linear program as described next.

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, SQi, SFCi and SSPi 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 SFC$_i$ models. The constrained energy optimization problem is formulated as a linear program as follows:

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

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:

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

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.

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

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

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

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

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

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

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

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

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

In the above inequalities, equations (5-3) through (5-5) capture the properties of a CTMDP. Inequalities (5-6), which are based on the Little's theorem [12], impose constraints on the expected task delay of SP$_i$, where $q_{i,x}$ represents the number of waiting tasks in the queue SQ$_i$ when the system is in state x, D_i is the expected service delay experienced by 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 in 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. It is our way of controlling the request blockage rate in the system. Constraint (5-8a) is imposed when each type of SR utilizes its own non-sharable SQ, whereas 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 mathematical program solver, i.e., MOSEK [13].

6 Experimental Results

For this experiment, we recorded a one-hour trace of device requests generated by four concurrently running applications on a Linux PC machine. The applications have two different types. Three of them are file manipulation programs, which read some data file, edit it and write back to the disk. The fourth application is 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 is modeled very well with a Poisson process with an average rate of 0.208 requests per second. The request generation statistics of the second program type may be characterized by a two-state CTMDP model, of which the state transition rate and the generation rates of SR to hard disk λ_{hd} and to WLAN card λ_{wlan} are $\begin{bmatrix} 0 & 0.0415 \\ 0.0063 & 0 \end{bmatrix}(s^{-1})$, $\begin{matrix}\lambda_{hd} = [0.0826, 0.0187] \\ \lambda_{wlan} = [0.1124, 0.1124]\end{matrix}(s^{-1})$.

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 the service providers. The power dissipation and start-up energy and latency of the disk drive and the WLAN card are reported in Table 1.

Table 1. Energy and transition data of hard disk driver and WLAN card

Hitachi 7K60	State	Power (w)	Start-up energy (J)	Wake-up time (s)
	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 really 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 within 3ms.

We used the lower envelope algorithm [14] as the timeout policy for the LPM, which was proven to be a 2-competitive policy for a device with multiple low power states. The LPM policy has 2 timeout values, each corresponding to one low power state. They are 1.7 s and 14.4s, respectively. 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. Again, 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. Simulation results of Hierarchal Power Management 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	1.0943	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 Power Management (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. It can be seen that HPM improves the energy efficiency of LPM controlled service provider, especially when there is a large positive slack with respect to the total delay constraint. HPM-S even outperforms the optimal component-level CTMDP policy.

In the second 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 set 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 3.

Table 3. Simulation results of Hierarchal Power Management 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

Experiment results demonstrate that the HPM-S algorithm can jointly schedule applications for different SPs to achieve minimal total energy.

7 Conclusion

This paper presented a hierarchical power management architecture which aims to facilitate power-awareness in an EMC system with multiple components. The proposed architecture divides power management function into two layers: system-level and component-level. The system-level power management was formulated as a concurrent service request flow regulation and application scheduling problem. Experimental results showed that a 25% reduction in the total system energy can be achieved compared to the optimal component-level DPM policy.

8 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, pp. 42–55, Mar. 1996.

[2] C-H. Hwang and A. Wu, "A predictive system shutdown method for energy saving of event-driven computation," 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, 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, pp. 1200-1217, Oct. 2001.

[5] Y-H. Lu, L. Benini and G. De Micheli, "Power-aware operating systems for interactive systems," *IEEE Trans. VLSI System*, Vol.10, pp. 119-134, Apr. 2002.

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

[7] T. Simunic, S. Boyd, and P. Glynn, "Managing Power in Networks on Chips", *IEEE Trans. VLSI System*, Vol.12, pp. 96-107, Jan. 2004.

[8] Z. Ren, B.H. Krogh, and R. Marculescu, "Hierarchical adaptive dynamic power management," *Proc. of DATE*, pp. 136-41, Feb. 2003.

[9] 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. DATE*, pp. 4–9., Feb. 2004.

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

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

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

[13] 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.

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