Dynamic Power Management Using Adaptive Learning Tree

Eui-Young Chung† Luca Benini‡ Giovanni De Micheli†
† eyehung,nanni}@stanford.edu
† ‡ lbenini@deis.unibo.it
Stanford University
Computer Systems Laboratory
Stanford, CA 94305-4070, USA

Dip. Informatica, Elettronica, Sistemistica
Univertà di Bologna
40136, Bologna, ITALY

Abstract

Dynamic Power Management (DPM) is a technique to reduce power consumption of electronic systems by selectively shutting down idle components. The quality of the shutdown control algorithm (power management policy) mostly depends on the knowledge of user behavior, which in many cases is initially unknown or non-stationary. For this reason, DPM policies should be capable of adapting to changes in user behavior. In this paper, we present a novel DPM scheme based on idle period clustering and adaptive learning trees. We also provide a design guide for applying our technique to components with multiple sleep states. Experimental results show that our technique outperforms other advanced DPM schemes as well as simple time-out policies. The proposed approach shows little deviation of efficiency for various workloads having different characteristics, while other policies show that their efficiency changes drastically depending on the trace data characteristics. Furthermore, experimental evidence indicates that our workload learning algorithm is stable and has fast convergence.

1 Introduction

The importance of system-level low-power design techniques has been increased by the widespread use of portable devices, which have limited battery life time [2, 4, 7]. Dynamic power management (DPM) [1] is a system-level low power design technique aiming at controlling performance and power levels of digital circuits and systems, by exploiting the idleness of their components. The heart of DPM is a Power Manager (PM) which monitors the overall system state and issues commands to control the power state of the system when it detects idleness. The control algorithm implemented by the PM is called a power management policy. Adaptivity is one of the most important issues in DPM because most external environments (user requests) are non-stationary.

Three classes of power management policies have been proposed in the past: time-out, predictive, and stochastic policies. The fixed time-out policy shuts down the system after a fixed amount of idle time [15]. Adaptive time-out policies are more efficient because they change the time-out according to the previous history. In contrast with time-out policies, predictive techniques do not wait for a time-out to expire, but shut down the system as soon as it becomes idle if they predict that the idle time will be long enough to amortize the cost of shutting down.

Some predictive techniques are based on extensive off-line analysis of usage traces [8]. Adaptive prediction policies [3] overcome this limitation by adopting an exponential average prediction scheme. Both time-out and predictive policies have been applied only to systems with a single sleep state. A stochastic approach was proposed in [5] where general systems and user requests were modeled as Markov chains. This approach provides a polynomial-time exact solution for the search of optimal power management policies under performance constraints. The main drawback of this approach is the assumption that the Markov model of the workload is stationary and known. This limitation is addressed in [6], where adaptive Markov policies are investigated. In [6], a look-up table is constructed which contains pre-optimized stationary policies. Decisions are obtained by interpolation on the look-up table. The table is indexed by workload parameters which are estimated online with a sliding window algorithm. The main limitation of this adaptive technique is that it is based on a fixed-time update rule that can be power-inefficient. Furthermore, in some cases, neither the workload nor the system can be modeled accurately by the Markov chains.

In this paper, we present a novel adaptive predictive method applicable to systems (or components) with an arbitrary number of sleep states. Our policy is based on a new dynamic data structure called an adaptive learning tree. Using the tree, we can accurately predict the most appropriate low-power sleep state at the start of an idle period. Also, we propose an enhanced scheme which adopts a time-out filter for the purpose of eliminating very short idle periods from being candidates for prediction with minor power penalty. We tested out the technique on the well-known power management problem of hard disk spindown. Our experiments, with different hard-disk drives and measured workloads, show that our adaptive technique is robust and it consistently outperforms time-outs and predictive policies, in terms of both power savings and performance penalty.

2 Idle Period Grouping

In this section, we introduce the idle period clustering scheme which is the base of our proposed DPM approach for a multi-sleep-state system. A system can be abstracted as a two-state finite-state machine which is in busy state when a service is performed and it is in idle state otherwise. An idle period is defined as the period from the time when the system enters the idle state to the time when the system exits the idle state. Similarly, busy periods are the time intervals spent in busy state. Thus, the overall system behavior can be modeled as a time series of busy
and idle periods. When an idle period is long enough to amortize the shutdown cost, the system can be shut down for power saving. For a system with a single sleep state, such as the one analyzed in [3] we can define a threshold, which is the minimum idle time required to reach the break-even point between shutdown cost and power savings. For a multiple sleep state system, we need as many thresholds as sleep states because each sleep state has a different shutdown cost. Figure 1 illustrates the need for multiple thresholds and the efficiency of multiple sleep states compared to the single sleep state when the system workload is known. Usually, shutting down to a deeper sleep state requires more transition time and power consumption (i.e., higher cost).

\[ I_i = \frac{(pd_{i+1} - p_i) \ast td_{i+1} + (pu_{i+1} - p_i) \ast tu_{i+1}}{p_i - p_{i+1}} \]  

(2)

The time axis can be partitioned in \( n + 1 \) disjoint intervals, bounded by the thresholds. We can then associate with a given idle period \( t_{idle} \) the index of the power state \( IG(t_{idle}) \) giving the best savings for that idle period:

\[ IG(t_{idle}) = \begin{cases} 
0 & \text{if } t_{idle} < t_0 \\
 i + 1 & \text{if } t_i < t_{idle} < t_{i+1} \text{ for } 0 \leq i < n \\
n & \text{if } t_n < t_{idle}
\end{cases} \]  

(3)

Thus, a sequence of idle periods can be transformed into a sequence of integers \( 0 \leq IG(t_{idle}) \leq n \), which represent the best power state that could be chosen for each idle period. Let \( s \) denote the sequence. If \( s \) has finite length \( l \), it is denoted as \( s^l \). Also, \( s_i \) denotes the \( i \)th value of the sequence and \( s_0 \) is the most recent event among all \( s_i \)'s. The optimal power state for an idle period represented by \( s_i \) is \( p_{s_i} \).

3 Adaptive Learning Tree

Predicting the values of a discrete event sequence is a fundamental problem in learning theory [11]. The idle period clustering technique mentioned in Section 2 transforms the sequence of idle periods into the sequence of discrete events. In other words, the problem to be solved is “which value will \( IG(t_{idle}) \) have in the next idle period for the current sequence \( s^l \)?” By predicting the next \( IG(t_{idle}) \), the system can choose the most appropriate sleep state. In previous studies, learning tree algorithms have been reported to find rules from experience [12, 13, 16, 17]. These algorithms are static in nature, and can be seen as techniques to organize knowledge and drive inference. To be effective, our algorithm must be highly dynamic, and be able to adapt rapidly to changes in the workload.

The learning tree that we propose can be applied to binary as well as multi-valued sequences. Idle periods are observed by the PM and they are transformed into integers \( IG(t_{idle}) \). This information can be seen as a sequence \( s^l \). The PM predicts the next \( IG(t_{idle}) \) for the given \( s^l \) based on the current status of the learning tree. The learning tree is updated as soon as the prediction result is available. \( s^l \) is updated by shift operation whenever a new idle period is observed by PM such that \( s_i \rightarrow s_{i+1} \) and the new value is stored as \( s_{i+1} \). The basic assumption behind our algorithm is that we can predict the future idle periods with high accuracy by observing idle periods in the recent past. Our approach has some analogy with advanced branch prediction schemes widely used in computer architectures to reduce the penalty of mispredicted branches [14].

3.1 Basic Structure

An example of an adaptive learning tree is shown in Figure 2. The proposed adaptive learning tree consists of decision nodes (circles), history branches (solid lines), prediction branches (dashed lines), and leaf nodes (rectangles). The tree is levelized: the top decision node corresponds to \( s_0 \), nodes in the second level correspond to \( s_1 \) and so on. All leaf nodes are predictions for the next idle period regardless of their ancestor levels. Each leaf stores the Prediction Confidence Level (PCL). The higher the PCL is, the higher the confidence is for a prediction.
Each decision node can have both history branches and prediction branches, but the total number of branches is always 2^i, and a prediction branch can only be used when the ancestor is a decision node and the descendant is a leaf node. Each branch of a decision node is associated with the index of a power state IGi(tidle) = {0, 1, \ldots, n}. From left to right, they are denoted as bsi, i = 0, 1, \ldots, n regardless of their types.

3.2 Decision
A decision for a given sequence, s' is taken based on a path matching procedure. A path for a given sequence, s' is defined as a series of decision nodes such that from the top node, we recursively select a history branch bsi and move to the lower level decision node connected to bsi. The recursion is terminated when the bsi is a prediction branch or the level of the decision node corresponds to s-i. Path length (pl) is defined as the number of decision nodes included in the path. While matching the path, the leaf nodes connected to the decision node included in the path are checked and the leaf node which has the highest PCL is selected. When there are multiple leaf nodes which have the same highest PCL, the leftmost leaf node is selected. After path matching, the index of the selected leaf node becomes the prediction for the next event. For example, in Figure 2, the path, “a \rightarrow b \rightarrow e” is matched when the s' = “01” and its path length is 2. After path matching, the center leaf node of node e is selected, thus the tree predicts IGi(tidle) = 1 for the next idle period and issues a command to shut down the system to power state 1. Also, when the s' = “00” or s' = “02”, the path, “a \rightarrow b” is matched. Note that node e is not included in the path for these sequences any more. Thus, the rightmost leaf node of node b is selected in this case. As shown in this example, pl, the number of old events used in decision is varied according to the given sequence. Also, two different sequences (“00” and “02”) are classified in the same category and can share the resources of the tree to reduce memory usage.

3.3 Learning
In conjunction with prediction, a learning process is needed to maintain the accuracy of the prediction. Whenever an event s1 occurs, the tree is updated to reflect the quality of prediction made when the previous event s1+i occurred. When the prediction is correct, the learning tree should be updated to increase the possibility to choose the same leaf node for the given sequence. In the reverse case, the reverse action should be performed. This task is achieved by updating the PCL of the leaf nodes. PCL update is controlled by a finite-state machine as shown in Figure 3 (the update rule is analogous to that employed in branch prediction buffers for conditional branch prediction). When the prediction is correct, the PCL state is changed to the higher state, in the reverse situation, the PCL state is changed to the lower state. And when it reaches either end state, it keeps the current state. Thus, the PCL is an adaptive feature of the learning tree for non-stationary event sequences. Learning process is more compli-

![Figure 2. An adaptive learning tree (with two sleep states)](image)

Figure 2. An adaptive learning tree (with two sleep states)

![Figure 3. PCL operation](image)

Figure 3. PCL operation

![Figure 4. An example of learning for a prediction miss](image)

Figure 4. An example of learning for a prediction miss

there are three different sequences such that A = “20”, B = “21” and C = “22” and the next event after sequence A and B is 0, but the next event after sequence C is 1. The path “a \rightarrow d” will be matched for all those sequences in Figure 4 (a) and the learning tree will predict 0 for every sequence. This prediction is correct when the given sequence is A or B; it is wrong when the given sequence is C. Thus, it is necessary to distinguish sequence C from A and B. When this prediction miss occurs, first, the PCL of the leftmost leaf node of node d is decreased. Then, the rightmost leaf node of node d is replaced with a new decision node because s1 = 2. The leaf nodes of the new decision node have the initial PCL value (in this case, it is 1). Then, the second additional procedure is applied and the final PCL of leaf nodes are as shown in Figure 4. Due to these additional procedures, the adaptive learning tree grows in an unbalanced manner and this characteristic is efficient for keeping it small and naturally determines the correlation depth between the future event and old history depending on the sequence characteristics.

4 Power Manager
As mentioned in Section 1, the Power Manager (PM) is the heart of DPM. Thus, the adaptive learning tree is implemented
within the PM as shown in Figure 5. In Figure 5, service requester

(SP) is the external environment which triggers the system and the service provider (SP) is the system itself which serves the requests from service requester. The idle period grouper (IPG) observes SP and extract idle periods. Then the idle interval for the observed idle period is calculated and it is passed to the previous history buffer (PHB) and the tree handler. PHB stores the observed idle sequence $s^i$. Whenever a new event arrives, it performs shift operation such that $s_j \rightarrow s_{j+1}$ and the new event is stored as $s_0$. Finally, the tree handler performs learning and the predictor performs the decision process as mentioned in Section 3.

**Wakeup and miss correction**

SP can be waken up in two different ways. One is when PM detects a new service request. The other is when SP stays in power state $p_i$ longer than $L_i$. The first case occurs when the predicted idle interval is greater or equal to the actual idle interval. And the second case occurs when the predicted idle interval is less than the actual idle interval. The second case is a prediction miss due to a conservative prediction. After SP is waken up, PM monitors the system until $L_0$ (maximum threshold). During this period, if a new service request comes, the SP can serve this request without wakeup penalty, thus the inefficiency in power saving is compensated by eliminating wake-up performance penalty. Otherwise, the PM shuts down the SP to the deepest power state to save more power. This feature enables the exploitation of very long mispredicted idle periods.

**Prediction filter**

In many applications, the distribution of idle period intervals shows L-shaped curve as mentioned in [3, 8] which represents the ratio of very short idle periods is dominant in total idle periods distribution. Thus, the prediction quality for short idle periods can play an important role in deciding overall prediction accuracy. For this reason, fixed time-out policy is performed preceding the actual prediction. In other words, the command predicted by PM is not issued immediately, but the command issue is delayed for a small amount of time (threshold of the fixed time-out policy) to filter out very short idle periods. If a request is arrived during this waiting period, the predicted command is canceled, thus only the idle periods longer than the threshold can be used for shutdown. The threshold value used for the fixed time-out policy is the minimum threshold, $L_0$. Usually, $L_0$ is small, thus the sacrifice to filter out short idle periods is not a big penalty for power saving, but it prevents excessively aggressive shutdown.

5 **Experimental Results**

We applied the proposed scheme to two different Hard Disk Drives [5, 10] with the real trace data [9]. We chose two different types of disk traces from [9] - one is the trace for swap purpose only disk and the other is the trace for swap and user data disk. Thus, the distributions of idle period length are different. Two different HDD specifications are shown in Table 1 with the threshold values computed by the equation 2. We implemented a simulator to estimate the performance of the proposed algorithm in terms of power consumption, delay overhead, and energy efficiency. Also, the simulator can simulate fixed time-out policies, the best oracle policy [6], and other predictive policies [3] for validation purpose. The best oracle policy is an ideal policy which cannot be implemented in practice because it assumes perfect knowledge of all idle periods, and it always takes the best decision. For the proposed approach, the size of PHB is determined 20 bits, thus the maximum path length of the adaptive learning tree was constrained to be less than or equal to 20. Since fixed time-out policy and the prediction policy in [3] does not support multiple sleep states, only the deepest sleep state is used for those policies.

The compared policies are: 1) best oracle (O1), 2) proposed approach without filter (M1), 3) proposed approach with filter (M2), 4) prediction policy in [3] with miss correction (H1), 5) H1 with pre-wakeup (H2), 6) time-out policy with time-out value = $L_0$ (T1), 7) time-out policy with time-out value = $1$ sec (T2), and 8) time-out policy with threshold value that is used in H1 (T3). O1 is the reference in comparison because any other shutdown technique cannot outperform O1 and H2 has the pre-wakeup feature in addition to the features of H1. Several quality measures as shown below were obtained from the simulation.

- **Hit ratio** (HR): is defined as the ratio between # of correct prediction to # of total prediction. Thus, it is not used for the fixed time-out policies. Also, the hit ratio of proposed approach can not be directly compared to that of [3] because they have different number of sleep states (unit: %).
- **Avg. power** (AP): is the average power consumption during SP is in idle state (unit: W).
- **Delay Overhead** (DO): is the ratio between the increased idle time after applying the policy and original idle time (unit: %).
- **Avg. delay / idle period** (AD): is the ratio between total increased idle time and total number of idle periods. It is a good quality measure for instant availability (unit: sec).
- **Energy** (EN): is the total energy consumed during idle periods normalized to the energy consumption by best oracle policy (unit: J).
- **Efficiency** (EF): is the ratio between the normalized energy in O1 and that of each policy. It represents well the efficiency of the policy compared to the ideal policy and good for considering the power saving and performance penalty together.

The simulation results are shown in Table 2. First, the effect of
Table 2. Comparisons of the various policies

<table>
<thead>
<tr>
<th>Policy</th>
<th>Efficiency</th>
<th>Energy</th>
<th>Hit ratio</th>
<th>Delay overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP</td>
<td>0.90</td>
<td>0.10</td>
<td>0.95</td>
<td>0.05</td>
</tr>
<tr>
<td>DO</td>
<td>0.80</td>
<td>0.20</td>
<td>0.85</td>
<td>0.15</td>
</tr>
<tr>
<td>AD</td>
<td>0.70</td>
<td>0.30</td>
<td>0.75</td>
<td>0.25</td>
</tr>
<tr>
<td>EN</td>
<td>0.60</td>
<td>0.40</td>
<td>0.65</td>
<td>0.35</td>
</tr>
</tbody>
</table>

Table 3. Comparisons for design guide

<table>
<thead>
<tr>
<th>Policy</th>
<th>Hit ratio</th>
<th>Delay overhead</th>
<th>Energy</th>
<th>Efficiency</th>
</tr>
</thead>
<tbody>
<tr>
<td>AP</td>
<td>0.90</td>
<td>0.10</td>
<td>0.10</td>
<td>0.90</td>
</tr>
<tr>
<td>DO</td>
<td>0.80</td>
<td>0.20</td>
<td>0.20</td>
<td>0.80</td>
</tr>
<tr>
<td>AD</td>
<td>0.70</td>
<td>0.30</td>
<td>0.30</td>
<td>0.70</td>
</tr>
<tr>
<td>EN</td>
<td>0.60</td>
<td>0.40</td>
<td>0.40</td>
<td>0.60</td>
</tr>
</tbody>
</table>

Figure 6. Distribution of Idle intervals

Figure 7. (a) Cumulative hit ratio (IBM HDD: trace data 0) (b) Cumulative hit ratio (IBM HDD: trace data 1)

Figure 8. Cumulative hit rate (Toshiba HDD: trace data 0)
shows choosing deeper intermediate sleep state (standby instead of idleLP) makes it possible to save more power. This situation depends on the distribution of idle intervals. In this experiment, deeper sleep states are preferred, because idle periods in both idle intervals 1 and 2 have similar ratios in the distribution. Nevertheless, the M2 of the second case shows better efficiency than the third case. The reason is that third case wastes more idle periods when filtering out impulse-like idle periods because its I0 is much larger than the I0 of the second case. Even though the hit ratio is increased by a large I0 (because of perfectly filtering out short idle periods), the increased ratio is only a small amount because the proposed approach already preserves high hit ratio. Thus, to adopt the proposed approach, choosing shallower sleep states or choosing deeper sleep state with small time-out value for filtering is recommended. And it is also shown that increasing the number of sleep states is not always the best choice, because it increases the difficulty of the decision process. The results of M2 from the first case and second case supports this argument because their efficiency is almost same and the hit ratio of the first case is lower than second case.

Finally, the proposed adaptive learning tree algorithm was implemented on the ACPI-compliant [18] Pentium II laptop computer with a Fujitsu MHL 2043 hard disk. The operating system running on the computer was a beta version of Microsoft Windows 5. The proposed algorithm was written in C language and easily ported on the computer thanks to the software controlled power management architecture introduced in [19]. Also, fixed time-out policy was implemented in the same environment for the comparison purpose. the threshold value used for the fixed time-out policy was 30 seconds as suggested in [20]. The workload trace used in this comparison was collected for two different users and its length was about 11 hours. As expected, the hit rate of the proposed algorithm was 94.5% for the given trace and the average power consumption while using the proposed algorithm was 8% less than that using the fixed time-out policy. But the delay overhead of the proposed algorithm was 0.5% larger than that of the fixed time-out policy. Notice that the hard disk used in this experiment provides only a single sleep state. We believe that both average power consumption and delay overhead of the proposed algorithm will outperform the fixed time-out policy by a larger margin when both algorithms are applied to the multiple sleep state hard disk.

6 Conclusion

In this paper, we presented a novel power management policy which is useful for multiple sleep state components. The proposed approach is based on an adaptive learning tree and idle period clustering, and it has been validated through extensive experiments using two different HDD models and two kinds of real disk trace data. The experimental results show that the proposed approach outperforms fixed time-out policy and other prediction methods. Also, it is shown that the prediction accuracy is reliable in the sense that the proposed approach is much less affected by strongly non-stationary workloads. Moreover, the proposed approach reaches reasonable hit ratio before experiencing more than 1000 idle periods. Finally, we implemented the proposed algorithm on a laptop computer with a power-manageable hard disk and showed its feasibility in a real system environment.

References


