Dynamic Power Optimization Targeting User Delays in Interactive Systems

Lin Zhong and Niraj K. Jha
Department of Electrical Engineering
Princeton University
Princeton, NJ 08544
{lzhong, jha}@princeton.edu

***Abstract*— Power has become a major concern for mobile computing systems such as laptops and handhelds, on which a significant fraction of software usage is interactive instead of compute-intensive. For interactive systems, an analysis shows that over 90% of system energy and time is spent waiting for user input. Such idle periods provide vast opportunities for dynamic power management (DPM) and voltage scaling (DVS) techniques to reduce system energy. In this work, we propose to utilize user interface information to predict user delays based on human-computer interaction history and theories from the field of psychology. We show that such a delay prediction can be combined with DPM/DVS for aggressive power optimization. We verify the effectiveness of our methodologies with usage traces collected on a personal digital assistant (PDA) and a system power model based on accurate measurements. Experiments show that using predicted user delays for DPM/DVS achieves an average of 21.9% system energy reduction with little sacrifice in user productivity or satisfaction.**

***Index Terms*— Energy efficiency, human-computer interaction, power management, user interfaces.**

I. Introduction

Power has become a dominant concern for mobile computing systems. While previous low power techniques were mostly concerned with compute-intensive applications, the usage of mobile computing systems is mostly interactive. With human users involved, the system behavior becomes very difficult to predict without user information, rendering many existing low power techniques much less effective.

Before the next user input, most interactive applications simply block and wait idly. Such applications are called *passively interactive*. For example, most personal information management (PIM) and office applications belong to this category. In other cases, the applications proceed in their own specific way if there is no user input, $e.g.$, in many gaming and global positioning system (GPS) applications. Such applications are called *actively interactive*. In this study, we focus on passively interactive applications and just use the term "interactive" to refer to them unless otherwise indicated. Our work was originally presented in [1] and is presented with significantly more details in this paper. To the best of our knowledge, our work is the first that addresses idle-time energy efficiency from the user's perspective, exploiting user interface information and human factors.

Acknowledgments: This work was supported by DARPA under contract no. DAAB07-02-C-P302.

A. Energy characteristics

We next present a small motivational example for the problem we wish to solve. Fig. 1 shows the system power consumption when a user interacts with the Qtopia [2] $Calculator$ on a Sharp Zaurus SL-5500 PDA to compute $(89 \times 56) \div 45$. The power is sampled 400 times per second. Details of the power measurement technique are presented in Section VIII. $Calculator$ is used in a manner similar to a real calculator. Its

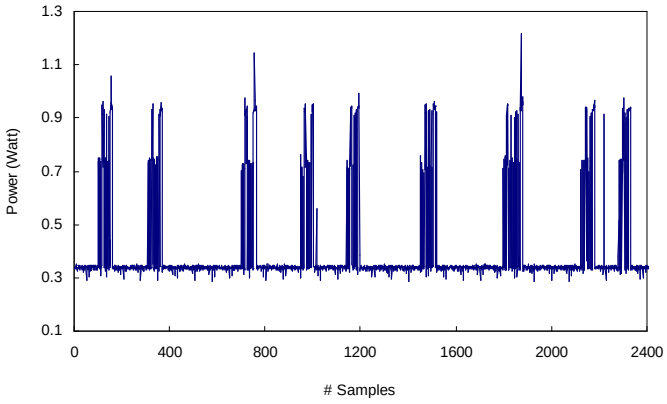


Fig. 1. Power consumption of Sharp Zaurus while running $Calculator$.

graphical user interface (GUI) presents buttons similar to the hardware buttons on a real calculator. The user makes inputs by tapping the GUI buttons with a stylus.

As shown in Fig. 1, there are valleys separated by nine major power peaks, which correspond to $Calculator$'s responses to tappings of GUI buttons. In the valleys, the system waits for user input while the operating system (OS) does maintenance jobs like handling timer interrupts and scheduling, which introduces small fluctuations and several minor spikes in the valley. Such power characteristics are typical of most interactive systems.

To see how much time and energy the idle valleys take, we analyzed usage traces from two users for four Qtopia applications shipped with the PDA (see Section VIII for

details). The analysis showed that over 90% of the time and energy was spent in waiting for user input. Similar findings have been made in [3] but for desktop computers. Moreover, most of the waiting periods are longer than 500ms. Such system idle periods are long enough for many systems to save energy by entering a low-power mode with a relative long exit latency, which is impossible for commonly used timeout-based power management. However, if the system is woken up from the low-power mode by user input, the exit latencies may be too long to be unnoticed by users, and may lead to severe degradation in user productivity. Therefore, to effectively utilize such opportunities and avoid sacrifice in computer responsiveness, two questions must be answered:

- When does the system begin waiting for user input?
- How long does it wait?

The answers to these questions also provide important information about how hardware resources will be used by the application. Therefore, they can be used to better drive dynamic voltage scaling and device power management as well. This work is primarily concerned with seeking answers for these questions.

B. Related work

A lot of research has been devoted to developing energy efficiency techniques for mobile computing systems. However, most past works focus on the system busy time instead of idle time, thus addressing the less important problems. For those that do address idle time, the techniques are preliminary.

DPM/DVS techniques are concerned with changing system performance levels dynamically to reduce system energy consumption, and have been extensively investigated [4], [5]. One of the key problems such techniques address is resource usage prediction. Various predictive and stochastic techniques have been proposed for this purpose [4], [6]–[9]. A number of techniques have been proposed to improve the prediction quality. For example, the power manager can be adapted to the resource usage history to make better predictions [4]. In these works, *time-out* is the common and implicit approach adopted for guessing when the system begins waiting for the next event. However, as demonstrated in [10], [11], most existing resource usage prediction methods may work well for compute-intensive tasks, but not for interactive tasks. In [12], an exponential cumulative distribution was used to model user requests for power management. This is stochastic in nature and cannot accurately predict individual user delays. More recently, application-specific information has been used for resource usage prediction [13], [14]. In [15], the authors proposed application programming interfaces (APIs) to utilize hints from applications for better I/O device power management.

Although many works [10], [11], [16], [17] have used interactive applications as their benchmarks, they have treated these applications in the same fashion as compute-intensive ones without exploiting any interactivity feature to make resource usage prediction better. Some of them [16], [17] have proposed to utilize the human-perceptual limit to slow down the system so that it can respond before the user can perceive the delay. However, they still focus on system busy time. In [18], the authors studied user interface event information for performing DVS. They also only targeted the system busy time. For GUI-based applications, we found in [19] that reducing GUI and display-related energy can be quite effective for interactive systems. It is also extremely important to accelerate application usage through GUI designs to achieve energy efficiency.

In this work, we will show how user delay prediction can be used to put the system into low-power modes with relatively long exit latencies during user delays without sacrificing computer responsiveness much. Using low-power modes with long exit latencies in a non-timeout fashion is by no means new. In [20], IBM researchers put a Linux watch system into its STANDBY mode whenever possible and let user input wake it up. The STANDBY mode had an exit latency longer than 200ms. Although such a latency was obviously perceptible to users and might not be acceptable for most applications on handheld computers, the authors considered it to be acceptable for applications on the Linux watch. In [21], the authors implemented a technique, called μSleep, that utilized the SLEEP mode of Intel SA-1100 with an exit latency of about 10ms to reduce system idle time power consumption. Since the latency is well below the human-perceptible threshold, there is no need for user delay prediction. However, the power consumption of the SLEEP mode of Intel SA-1100 can be further reduced by disabling the 3.6864MHz clock but with an exit latency of about 160ms. In this case, user delay prediction is necessary to guarantee computer responsiveness as we will discuss later on. However, neither work addressed user delay prediction for power management.

Like most DPM/DVS methods that use application information, our approach also suffers from the limitation that it requires changes to be made to the application source code, especially the user interface, or GUI toolkits. However, in view of the limited number of GUI toolkits for popular handheld computers and smart-phones, we believe a GUI toolkit based implementation does not impose a major burden.

C. Paper contributions and overview

As is evident from above discussions, new approaches are needed to answer the questions posed in Section I-A to exploit the energy-saving opportunities. We outline our approaches next.

1) When does the system begin waiting for user input: Most interactive applications are implemented in an event-driven fashion with an eventloop and a number of event handlers. When the system is in the eventloop, it is idle, waiting for user input. Upon receiving a user input, the system calls the corresponding event handlers to respond. After the event handlers finish execution, the system returns to the eventloop. This is illustrated in Fig. 2 using the Qtopia $Calculator$ as an example. Note that only the two primary event handlers, *enterNumber(int n)* and *std_buttons(int n)*, are shown for clarity. They are invoked by user tapping a digit button and a functional button, respectively.

The system starts waiting *whenever it returns to the eventloop*. Therefore, the application can simply notify the system

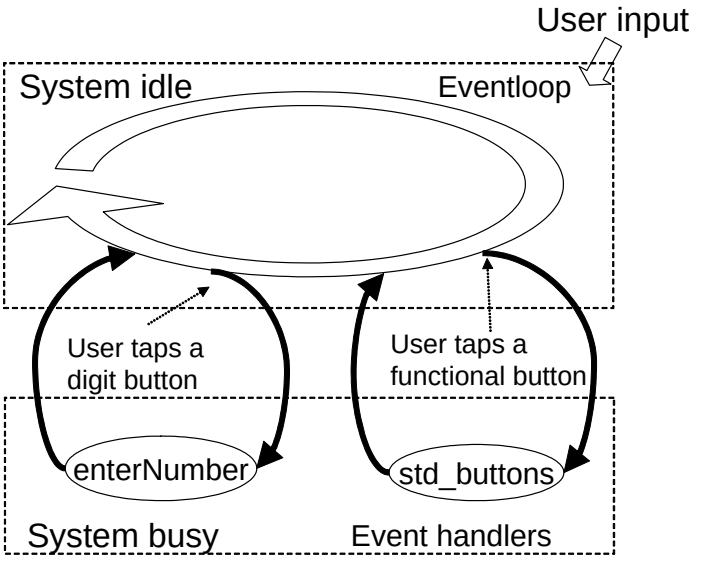


Fig. 2. Interactive application: eventloop and event handlers

when it enters the eventloop.

2) How long does the system wait: This question is equivalent to how long the user delay is. Without the knowledge of what the user is responding to, it is extremely difficult to answer this question. This is exactly why most conventional DPM/DVS policies are not much successful for interactive systems. Any information from the interactive application will help. If we know the application type, we may be able to estimate user delays better since, for example, user delays for text typing are statistically much shorter than those for playing *Go*. We can keep a user delay record for each application and predict its next user delay based on the application-specific record. This will improve the predictability of user delays, yet still suffer from the fact that the user may incur very different delays in different parts of the same application. For example, the user delay after clicking the "File" menu in a text editor will be much longer than that after keying in a letter.

A step further from the above monolithic view of an application would be to make a distinction between different parts of an application that have a different impact on user delays. We propose to model or partition the application from the user's perspective. Each partition, called *state* in the following discussion, presents relatively consistent information to the user, and requires relatively similar user reactions. Therefore, such a state will have more consistent and predictable user delays than the whole application. We address how an interactive application can be modeled as a finite-state machine for this purpose in Section II. Based on such a modeling, we propose to utilize user interface information and theories from psychological studies to predict each state's user delay lower bound. Without utilizing user interface information, we can also predict the next user delay in a state based on the history of user delays in that state. These two prediction approaches are described in Sections III and IV, respectively.

Based on our approaches to address the two questions posed earlier, we show how the proposed user prediction techniques can be implemented in Section V. The predicted user delays can be combined with DPM/DVS techniques to significantly reduce system energy consumption. A theoretical analysis for user delay prediction-based DPM/DVS policies is offered in Section VI. The energy benefits of our techniques are demonstrated based on a StrongARM-based Linux PDA. The usage traces, PDA system power model and its DPM/DVS policies are detailed in Section VII. The benchmarks and experimental setup are described in Section VIII, and experimental results in Section IX. Discussions are offered in Section X and conclusions in Section XI.

II. Application modeling from the user's perspective

In this section, we explain how an interactive application can be partitioned into states so that what the user sees and how he/she responds is relatively consistent in any given state, making the state user delay more predictable.

A. State-transition diagram extraction

State-transition diagrams (STDs) [22] have been proposed earlier to model computer-human interaction and specify user interface design. In this approach, an interactive application is modeled as a finite-state machine. Each state corresponds to a unique functional stage that waits for user input. The application changes state upon a user input/action. Such an STD, provided as a specification of the application, offers a design guide. However, we need an STD extracted from the implementation in order to implement user delay prediction.

As illustrated in Fig. 2, interactive applications are usually implemented with an eventloop and event handlers, which define an STD. The application waits for user input in some specific "state". Since an event handler processes user inputs and generates new information for the user, it transitions the application into another state (in some cases, the transition may be to the same state). Therefore, event handlers, with their triggering user inputs/events, correspond to transition arcs of the STD. If we assume every event handler transitions the application into a unique state, an STD is naturally defined by the event handlers. The event handlers that lead to a state in the STD are called *state triggers*. The event handlers that lead the application out of a state are called *state escapers*.

As an example, the STD for the Qtopia $Calculator$ is extracted as shown in Fig. 3. There are two states: $Number$ after a digit button is tapped and $Function$ after a function button is tapped. The corresponding state triggers are *enterNumber(int n)* and *std_buttons(int n)*, respectively. These two event handlers are state escapers for both $Number$ and $Function$. Such a $complete$ STD is actually the most detailed STD that can be extracted from the implementation. It provides the complete system-user interaction description. Note that an event handler in the implementation may correspond to more than one transition arc since the same event handler can be activated from different states. Therefore, all the arcs entering a state correspond to the same event handler. In the complete

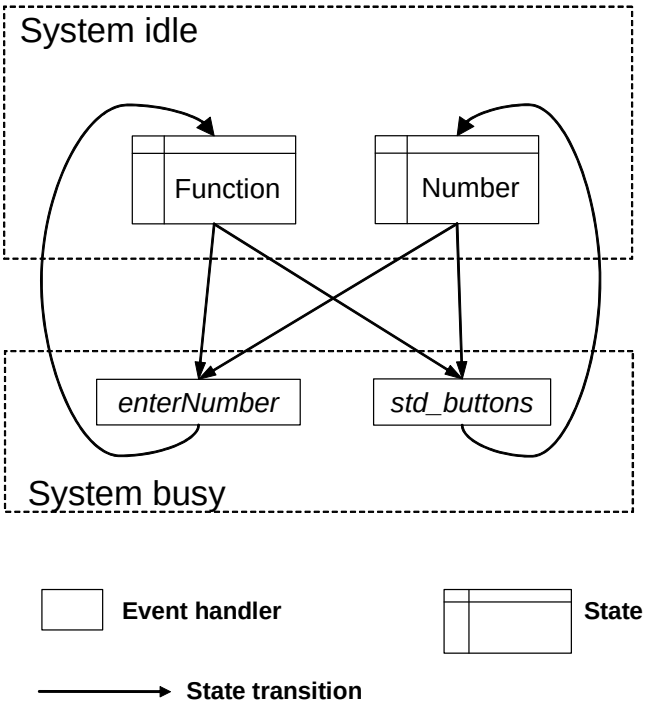


Fig. 3. The STD for the Qtopia $Calculator$.

STD, there is a one-to-one mapping between the set of states and the set of event handlers. If we only care about some specific user delays during system-user interaction or it is hard to derive the complete STD, we can identify event handlers which lead to the important states only. Instead of a complete STD, we will then have a $partial$ STD, in which there is a one-to-one mapping between the set of states and a subset of event handlers. Moreover, an STD, whether complete or partial, can be compacted by clustering multiple similar states into one, based on user interface content or profiling, to form a *clustered* STD. In a clustered STD, a state may correspond to more than one event handler.

The event handler-based STD provides us an opportunity to implement state-specific user delay prediction. Once we know that a set of event handlers will lead to a state, we can insert code into these event handlers to convey to the power manager the identity of the state and the predicted user delay. This is addressed in Section V.

B. User delay prediction

Unlike many other prediction problems, errors in user delay prediction may have very different effects on the system and the user. In other words, the cost of errors is heterogeneous, making solutions from standard time-series prediction theories [23] inappropriate. If the predicted delay is shorter than the real user delay, the prediction is an underestimation and conservative. If it is longer than the real user delay, it is an overestimation and aggressive. Underestimation wastes power-saving opportunities and overestimation may lead to user-perceptible latencies if the system is put into a power-saving mode with a long exit latency. It is well-known in human-computer interaction research that a longer system response time may lead to user productivity degradation and such a degradation is dependent upon users and tasks [24]. For example, it was demonstrated in [25] that system response latencies beyond 1 second could degrade user productivity by about 50% for some tasks. Such a degradation may yield back any gain in system energy efficiency through the use of power-saving modes. Unfortunately, there are no established direct quantitative links between system energy efficiency and user-perceived latencies. In this study, we will evaluate prediction methods in terms of the tradeoffs between energy savings and user-perceptible latencies they can make. One method can be better than another only in the sense of Pareto optimality.

III. User delay models based on psychological studies

If the content of the user interface of an STD state is known, the length of the user delay can be predicted based on what a user sees and how he/she needs to respond. There are three basic processes involved in a user's response to a system [26]. During the *perceptual* process, the user senses inputs from the physical world. In our scenario, it is the process of the user reading information presented on the display. During the *cognitive* process, the user decides on a course of action based on the input. In our scenario, it is the process of the user deciding which button to push, which menu item to click, $etc.$ Finally, during the *motor* process, the user executes the decision by mechanical movements. These three processes are consecutive in time. Before the user physically touches the system input peripherals, the system is idle.

A. Perceptual delay

For most modern computer systems, users get information from the system through visual and auditory channels by reading and listening. In this study, we focus on the visual channel.

Psychological studies have shown that humans read through discrete eye movements, which consist of $fixations$ and $saccades$. Fixations are states in which the eyes are focused on a object statically. Saccades represent rapid eye movements from one fixation location to another. Information is absorbed only in fixation, which lasts from 60 to $700ms$. A saccade takes about $30ms$. To estimate the perceptual delay, we have to estimate the number of fixations (or the number of saccades) and the duration of each. In order not to negatively impact user productivity, we estimate durations of these delays conservatively.

Computer display information can be categorized as graphical and textual. Graphical information is presented through graphical objects which consist of 1) window objects, such as buttons, radiobuttons, menus, menu items, $etc.$, and 2) pictures or figures. To simplify delay estimation, we assume each graphical object requires one separate fixation and each fixation lasts for the minimal $60ms$. Therefore, for each graphical object, a delay of $90ms$ (one minimal fixation plus one saccade) is added.

For textual information, psychological studies [27] have shown that college students can raud (read with comprehension) at a typical rate of 300 standard-length words per minute

or five per second. The standard-length word was assumed to have six characters. For a text with n words, the typical time for rauding is

$$T = \frac{n \times cpw}{Wpm \times 6} = \frac{n \times cpw}{30} \quad (s)$$

where cpw is the average number of characters per word and Wpm is the rauding rate in terms of the number of standard-length words per minute. If a text is associated with a graphical object, the reading time is used as the fixation length for the latter. For example, the delay to get information from a message box which contains a four-word text with an average of six characters per word, and two buttons with "OK" and "Cancel" on them, respectively, can be estimated as

$$T = 0.03 + \frac{4 \times 6}{30} + 0.03 + \frac{1 \times 2}{30} + 0.03 + \frac{1 \times 6}{30} = 1.157 \quad (s)$$

where there are three saccades and three fixations, whose lengths are estimated using the rauding rate.

B. Cognitive delay

The Hick-Hyman Law [28], [29] states that the time to make a decision (the reaction time, or RT) based on N distinct choices of equal probabilities is given by

$$RT = a + b \log_2 N \quad (s) \tag{1}$$

Parameters a and b are constants that can be empirically determined. Based on the information in [26], we assume a to be 0 and b to be $\frac{1}{7}$. Therefore, to decide which one from two buttons to tap, a user would think for approximately 0.14 second. To choose one item from a menu with four different items, a user would think for approximately 0.29 second. Table I summarizes how we apply the Hick-Hyman Law to various user interface features.

TABLE I
NUMBER OF CHOICES FOR DIFFERENT USER INTERFACE FEATURES USED IN THE HICK-HYMAN LAW

GUI features	number of choices
Group of N buttons	N
Group of N radiobuttons	N
Each checkbox	2 (on or off)
Menu bar with N items	N
A list of N items	N

The Hick-Hyman Law can be useful only if the number of different choices, N, is known. Unfortunately, it is hard or even impossible to estimate N for many cognitive processes. For example, menu selection may actually involve more choices for a user than the number of menu items because the user may evaluate the consequence of selecting an item and the possible subsequent choices to decide whether to select that item or not. A good Go player evaluates the current choice by considering what choices he or she would have many steps after this pending move. In this study, we conservatively assume N to be the number of most direct choices.

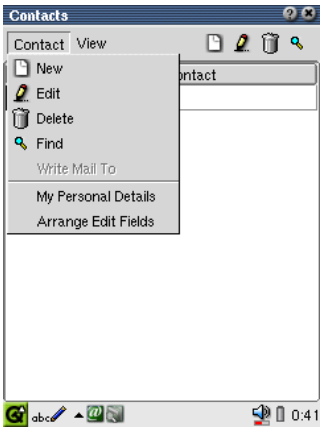


Fig. 4. A menu window from the Qtopia application $Contacts$.

C. Motor delay

It also takes time for a user to make a physical movement. In most PDAs, users respond by touching virtual objects on the touch-screen with a stylus. The Fitts Law [30], [31] states that for a normal human being, the motor time (MT) to carry out a movement is related to the size of the target and the distance to the target as follows:

$$MT = 0.23 + 0.166 \cdot \log_2(\frac{A}{W} + 1) \quad (s) \tag{2}$$

where A is the amplitude of the movement and W the width of the target as measured in the direction of motion. We have adopted the values for coefficients from [31].

The above three models are together referred to as *the psychological model* in this work. As an example, consider the popup menu window shown in Fig. 4 with four items. It will take four fixations and four saccades for the user to examine the menu. The perceptual process will take about 360ms. The Hick-Hyman Law indicates that about 290ms will be needed for making a decision from the four choices. Suppose the hand movement altitude is about one quarter of the screen height and the width of the target (the menu item) is about one sixteenth of the screen height. Then the Fitts Law indicates that the motor time will be about 600ms. Altogether, this GUI state will take an average user more than one second to respond to. User delays of such duration are long enough to put the system into a low-power mode with an exit latency up to several hundred milliseconds. However, they are still too short for conventional timeout-based power management policies.

Perceptual and motor delays can be predicted relatively well. However, the cognitive delay can be only conservatively predicted based on the Hick-Hyman Law, as mentioned before. Therefore, the psychological model is used to predict the lower bound on the user delay in this study. In the next section, we propose a history-based user delay model to predict the actual delay.

IV. HISTORY-BASED USER DELAY MODEL

The psychological model is user interface content-based. It predicts user delay based on what users will see. In some

cases, it may be difficult to know the user interface content. Moreover, it only offers one tradeoff between energy savings and user-perceptible latencies, and does not adapt to different users and the same user at different times. Therefore, we also devise a user delay model based on past observations. Before introducing the model, we first look into the statistics of user delays.

A. User delay statistics

Fig. 5(a) shows the trace of user delays for the state *Number* of the Qtopia $Calculator$, collected over three days. Note that the X-axis is not time. The lower bound on user delays predicted by the psychological model is plotted in Fig. 5(a) and also shown in Fig. 5(b).

As illustrated by Fig. 5, the user delays for a state are quite randomly but tightly centered around the mean at which the major peak in the distribution is located. A small group of randomly-occurring long delays form a minor peak in the distribution too. Such randomness makes accurate user delay prediction impossible. Fig. 5(a) also shows the learning effect over the three days: the user delay, in general, declines slightly, however, in a not very obvious fashion. We also observed that the standard deviation of user delays in a state is large when the corresponding mean is large.

B. History-based model

If we have gathered a long-term history of user delays of a state, we can obtain a distribution similar to Fig. 5(b). By using values at different percentiles of the sorted long-term history, we can make different tradeoffs between overestimation and underestimation, leading to different tradeoffs between energy savings and user-perceptible latencies. For example, if we use the minimal value or the value at a very low, say fifth, percentile of the long-term history, we can essentially achieve a lower-bound prediction even better than the prediction based on the psychological model. There are, however, some limitations of such methods. First, they are not reliable until a significant history record is available. For example, using the minimal value in the history record can lead to a lot of overestimation before a reasonable short user delay is observed. Second, they do not capture short-term trends effectively since such trends may be easily obscured in the long-term history. The tradeoff between energy savings and user-perceptible latencies may undergo unwanted changes over time. Therefore, although long-term history-based methods have some value, short-term history-based methods are preferred since they do not require a significant history and can adapt to trends efficiently. Moreover, short-term history-based methods are computationally much simpler in general.

Based on the above comments, we devise a prediction method similar to median filtering based on recent history. Let D_s^i denote the ith last observed delay for an STD state s and N the total number of past user delays recorded for that state. The larger N is, the longer is the history used.

The delay record is sorted to generate a new series so that

$$D_s^1 < D_s^2 < ... < D_s^N$$

Then we define the *habitual set* $\Phi(p)$ as

$$\Phi(p) = \{D_s^{\lfloor N \cdot p \rfloor + 1}, D_s^{\lfloor N \cdot p \rfloor + 2}, ..., D_s^{\lfloor N \cdot (1-p) \rfloor}\}$$

where $0 \leq p \leq 0.5$ with a typical value of 0.25. $\Phi(p)$ contains the central $N \cdot (1 - 2 \cdot p)$ items of the sorted series. It is a set extension of the concept of a median. When $p = 0.5$, it reduces to the median. When $p = 0$, it leads to the mean instead. Let m_Φ denote the mean of $\Phi(p)$. Then the next user delay for the state is predicted as

$$D = \beta \cdot m_\Phi$$

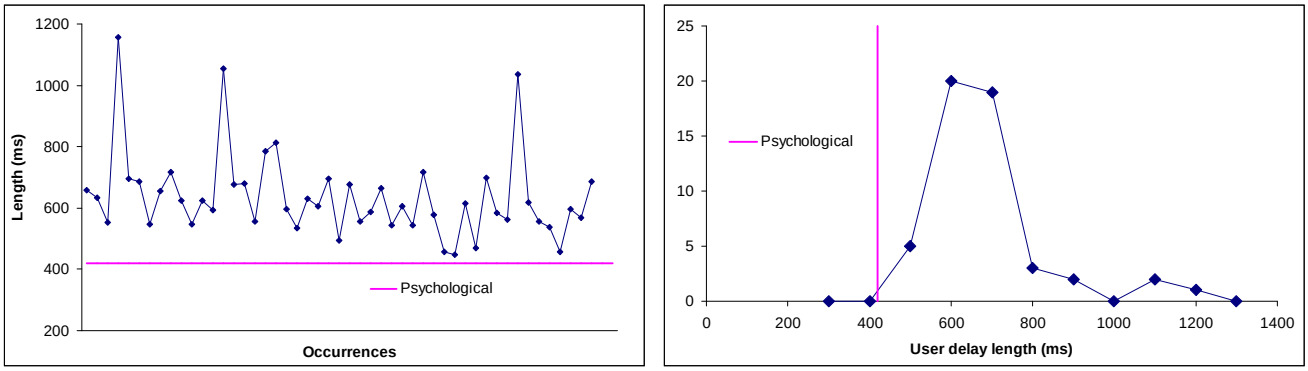
where $\beta > 0$. β is called the $pessimism$ factor and is used to generate different tradeoffs between overestimation and underestimation. Note that the mean instead of a percentile value of the habitual set is used since N is usually small. There are two rationales behind using a multiplicative factor instead of an additive one. First, different states have different mean user delays, and second, states with larger means usually have larger standard deviations. A constant multiplicative factor works better for different states than a constant additive factor.

C. History-based vs. psychological models

The history-based and psychological models are intended for different purposes. The psychological model is intended to predict the lower bound on the user delay of a state. On the other hand, the history-based model is used to predict the true length of the next user delay. By using the pessimism factor to scale the prediction, it can achieve various tradeoffs between energy savings and user-perceptible latencies. The psychological model is intended to predict the lower bound as a constant that holds for all users and even after enough practice. The history-based model is intended to adapt to different users and the same users at different times effectively.

The history-based and psychological models complement each other. Indeed, a history-based model using the minimal value or the value at a small percentile of the history can achieve the same goal as the psychological model for predicting the lower bound on the user delay, but after enough observations have been made. It is useful only when a long history is available for the user and initial errors of overestimation can be tolerated. In the case that overestimation is not tolerable, the psychological model can be used before a long history is available.

The history-based model enjoys simplicity of implementation. No knowledge about the user interface content is required. On the other hand, the psychological model has to know the user interface content, which may be complex and dynamic. Moreover, the Hick-Hyman Law has limitations in modeling complex cognitive processes. Thus, the psychological model is more accurate for user interfaces that do not involve complicated cognitive processes. Therefore, different delay models should be used for different states and different applications to exploit the strengths of both the history-based and psychological models.



(a) User delay trace

(b) User delay distribution

Fig. 5. User delay statistics for State $Number$ of $Calculator$ for user 1.

V. Implementation of User Delay Prediction

In this section, we address how user delay prediction can be implemented based on the application source code and GUI toolkit. We assume the application can convey to the power manager the predicted user delay, D, via a functional call $NotifyDelay(D)$. The power manager then determines what step to take for saving energy, which will be addressed in Section VII.

A. Fully customized implementation

When the application source code is available, one can extract the STD for important states and augment the corresponding event handlers for user delay prediction. The beginning of a state's user delay is marked by the end of its triggers, where $NotifyDelay()$ should be inserted. For history-based prediction, a global variable, $current\_state$, is added to note the current state. It is updated at the end of state triggers. A global array of data structures is added for maintaining a delay record for each state. The array is indexed by $current\_state$.

The arrival of the next event, or the start of a state escaper, marks the end of a user delay. Code can be inserted into the eventloop or corresponding state escapers to estimate the real user delay and update the state's user delay history.

In the $Calculator$ example, the start and end of event handlers *enterNumber(int n)* and *std_buttons(int n)* mark the end and start of user delays, respectively. At their start, the last user delay is estimated and used to update the delay record according to $current\_state$. At the end, $current\_state$ is updated, the starting time of a user delay is marked, and $NotifyDelay()$ is inserted.

B. GUI toolkit-based implementation

GUI applications are usually built using GUI toolkits, which provide implementations of the eventloop mechanism, GUI objects and their corresponding event handlers.

The proposed user delay prediction mechanism can actually be incorporated into the GUI toolkit so that any application built with it will have user delay prediction capability. In this subsection, we demonstrate how this can be implemented within the Qt GUI toolkit [32] using the example of popup menu windows like the one shown in Fig. 4. In Qt, popup menus are instantiated as objects of the class *QPopupMenu*. *Qpopupmenu::show()* and *Qpopupmenu::hide()* are called whenever a popup menu window is shown and hidden, respectively. The beginning of the user delay is marked by the end of $show()$, into which $NotifyDelay()$ can be inserted.

For history-based prediction, the user delay can be estimated as the time elapsed between the end of $show()$ and the beginning of $hide()$ by augmenting $show()$ and $hide()$. The data structures that maintain the history can be added to the $QPopupMenu$ class. They are accessed at the end of $show()$ to make the prediction for $NotifyDelay()$ and updated at the beginning of $hide()$ with the newly observed used delay.

The *QPopupMenu* class also provides enough data through member methods for psychology-based prediction. $size()$ returns the size information for the popup window. *count()* and *text(int id)* return the number of items on the menu window and the text for the idth item, respectively. A psychology-based prediction can be obtained with such information.

VI. Theoretical Analysis of DPM/DVS for Interactive Systems

In the previous sections, we discussed how to determine when a user delay occurs and how to predict its length. We also demonstrated how user delay prediction can be implemented based on application source code and GUI toolkit. With the $NotifyDelay(D)$ API, an interactive application can convey the predicted user delay to the power manager. We next show how user delays can be used to reduce system energy consumption through straightforward DPM/DVS policies.

A. DPM and DVS techniques

DPM/DVS techniques change the system performance level to save power while trying to meet soft/hard deadlines. The performance level of an interactive system can be changed by shutting down idle components, putting the processor into different power modes, and scaling the frequency and supply voltage. Most modern processors used in mobile computing systems support performance level changes through software, $e.g.$, Intel StrongARM and XScale processors [33]. Different

performance levels have different power consumptions. Transitions among levels cause delay and energy consumption. This must be considered when performing DPM/DVS.

Since many GUI operations can be quite demanding, specially on PDAs, the ideal setting would be the lowest possible performance level during user idle time and an appropriate higher level during system response.

B. Theoretical energy saving and latency overhead

We next present a theoretical analysis of the energy saving and latency overhead when DPM/DVS techniques are combined with user delay prediction.

For simplicity of exposition, we consider two performance levels for a given user delay. Let D denote the length of the user delay, P the system power consumption for the higher performance level, and P' that for the lower one. The energy consumption for the system during the user delay without DPM/DVS will be

$$E = D{\cdot}P$$

1) Perfect prediction: Suppose we can predict the user delay perfectly, change the system performance to the lower one after the system finishes responding, and change it back to the higher one right before the next user input. There is no latency overhead. Let P_{HL} and P_{LH} denote the performance level transition power, and T_{HL} and T_{LH} denote the delay for higher-to-lower and lower-to-higher transitions, respectively. We assume that $D > (T_{HL} + T_{LH})$. Then the energy consumption for the system during the user delay with such a performance-level transition is given by

$$E_0 = (D - T_{HL} - T_{LH}){\cdot}P' + T_{HL}{\cdot}P_{HL} + T_{LH}{\cdot}P_{LH}$$

If we assume $P = P_{HL} = P_{LH}$, the energy saving ratio is therefore

$$\rho_0 = \frac{E - E_0}{E} = \{1 - \frac{(T_{HL} + T_{LH})}{D}\} \cdot (1 - \frac{P'}{P}) \qquad (3)$$

If the system changes the performance level back to the higher level upon a user input, we can obtain the energy saving through a similar analysis as

$$\rho' = \rho_0 - \frac{T_{LH}}{D} \cdot \frac{P'}{P} \qquad (4)$$

In this scenario, the latency overhead is T_{LH}, which can be larger than the 50-100ms human-perceptible threshold.

2) Imperfect prediction: If the user delay models are used to predict D as D' and the system is put into the low performance level if $D' > (T_{HL}+T_{LH})$ and put back into the high performance level right before the predicted user delay elapses, we have

$$\rho = \begin{cases} \rho_0 - \frac{T_{LH}}{D} \cdot \frac{P'}{P}, & D' \geq D + T_{LH}; \\ \rho_0 - \frac{|D-D'|}{D} \cdot \frac{P'}{P}, & D \leq D' < D + T_{LH}; \\ \rho_0 - \frac{|D-D'|}{D} \cdot (1 - \frac{P'}{P}), & (T_{HL} + T_{LH}) \leq D' < D; \\ 0, & D' \leq (T_{HL} + T_{LH}). \end{cases} \qquad (5)$$

The latency overheads for these four situations are T_{LH}, $|D' - D|$, 0, and 0, respectively.

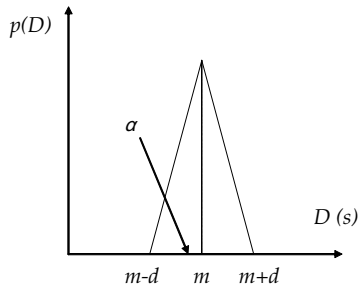


Fig. 6. Triangle distribution of user delays for a state

It is obvious that the energy saving ratio will increase linearly if P'/P decreases. Given the two performance levels, the energy saving ratio is only dependent on the user delay and prediction error. Comparing ρ with ρ', we can see that using predicted user delay is actually more energy-efficient if T_{LH} is large compared with user delay prediction errors. Note that both ρ and ρ' can be negative, which means energy consumption may increase if performance-level transition is not properly done.

When $D' > D$, the performance level transition will be triggered by a user input, resulting in a delay between $D' - D$ to T_{LH}. This is called a *lazy error*. When T_{LH} is larger than the human-perceptible threshold, H, employing user delay prediction for DPM/DVS may introduce user-perceptible latencies, leading to what is called a *serious lazy error* (such errors are discussed in greater detail in Section IX). The user-perceptible latencies also include the system busy time to respond. According to our analysis of the usage trace, system busy times are extremely short, usually much less than 5ms, making negligible contributions toward user-perceptible latencies. Therefore, we ignore them in the following analysis and in our experiments. To derive the percentage of serious lazy errors (PSLE) out of all system responses to user inputs, we assume the user delay, D, for a state has a triangle-distribution, $p(D)$, as shown in Fig. 6. Such a distribution is a reasonable approximation to that shown in Fig. 5(b). The mean, m, denotes how fast the user is in responding to this state, while the width, d, denotes how predictable the user is. Note that the standard deviation is $\sqrt{2/3}{\cdot}d$. If $\alpha \leq m$ is used to predict the user delay, the PSLE will be

$$PSLE = 100\% \cdot \int_0^{\alpha - H} p(D)dD$$

$$= \begin{cases} 0\%, & \alpha < m + H - d; \\ 50\% \cdot [1 - (\frac{m - \alpha + H}{d})^2], & m > \alpha \geq m + H - d. \end{cases}$$

It is clear that PLSE increases linearly as the prediction α approaches the mean m. It decreases linearly as the mean increases, demonstrating longer delays have fewer serious lazy errors. It also decreases as d decreases, showing better user delay predictability also reduces serious lazy errors.

VII. Power models and DPM/DVS Policies

Based on a predicted delay, the power manager selects an appropriate power-saving performance level for the state. The system returns to the higher performance level either after a

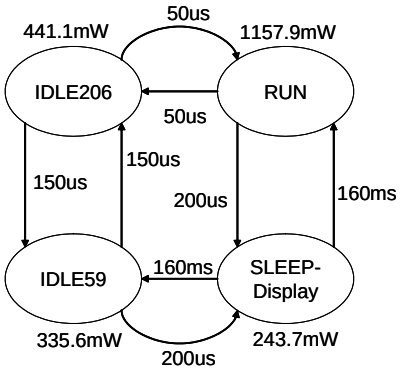


Fig. 7. Power modes and mode transitions.

time interval based on the predicted user delay, or just upon the next user input. Which of these two scenarios is targeted depends on how aggressive dynamic power optimization needs to be. For example, the delay overhead for performance level transitions for DVS techniques is typically less than 1ms [11], [13], [33]. Thus, one could just let user inputs trigger performance level transitions. However, for more aggressive power management with an attendant increase in the delay overhead, it is often necessary to raise the performance level for prompt system response before the user input arrives. For example, the delay for an Intel StrongARM SA-1110 to transition from the SLEEP mode to the RUN mode is about 160ms [33]. In such cases, the predicted user delay can be used. In this section, we first establish a system power model for the StrongARM-based Linux PDA used in our experiments. Then we discuss possible DPM/DVS policies for it.

A. System power model

The Sharp Zaurus SL-5500 PDA has an Intel StrongARM SA-1110 processor [33], which can run in three modes: RUN, IDLE, and SLEEP. Moreover, its core clock speed can be varied between 59 and 206MHz in discrete steps. It is put into the IDLE mode by the Linux kernel whenever there is nothing to run. We constructed a system power model for the PDA based on power measurements and available industrial data. The modes of system operation and their corresponding power consumptions are shown in Fig. 7. Permitted mode transitions are shown as directed arcs. The delay overhead for mode transitions is marked on the directed arcs [33]. We assume that the front-light is always off and the display is always on with constant power consumption of 234.4mW. We assume the power consumption is constant for each operation mode. Notably, the display consumes about half of the system idle power.

In the system power model, RUN corresponds to the SA-1110 RUN mode when the processor is busy executing instructions. Its power is measured when the PDA computes discrete cosine transforms, as detailed in [19]. IDLE206 and IDLE59 correspond to the SA-1110 IDLE mode with a core clock frequency/voltage of 206MHz/1.5V and 59MHz/1.25V, respectively. IDLE59 is a hypothetical mode for the Zaurus SL-5500 based on other SA-1100 or SA-1110 systems [11], [13], [34]. It is estimated as the display power plus the measured non-display power at 206MHz scaled down using the same ratio of the power for IDLE59 to that for IDLE206 in the Itsy system power measurement presented in [34]. SLEEP-Display corresponds to the SA-1110 SLEEP mode when the display is left on. Therefore, its system power consumption is measured when the system is suspended, and the display power is added thereafter. Compared to IDLE59, the SLEEP-Display mode saves more power. However, it should be used with caution since the transition time back to the RUN mode is large enough for a user to notice.

The power measurement equipment consists of a Windows PC, a hardware data acquisition card, and a wire connector box. The PC is installed with *National Instrument*'s data acquisition software called $Labview$ [35]. The voltage is measured across a sense resistor connected in series with the battery to obtain the system power consumption.

B. DPM/DVS policies

With the user delay information as obtained using models presented in the previous sections, there are different ways to apply DPM/DVS based on the system power model. The following DPM/DVS policies are considered in this study.

1) Linux policy: The Linux kernel automatically puts the system into the IDLE209 mode whenever there is no process running and returns it back to the RUN mode upon interrupt. We take this as the baseline and report results for other techniques against it. The Qtopia environment is also able to power-manage the front-light and system based on timeout. However, such timeout-based policies do not benefit from user delays, which are usually no more than a couple of seconds.

2) Simple and lazy policies: The most straightforward DPM/DVS policy would be to put the system into the IDLE59 or the SLEEP-Display mode right after the system finishes responding to the user and put it back into the RUN mode upon a user input. These policies are called the $simple$ and $lazy$ policies, respectively. Their energy savings can be computed using Equation (4). Since the IDLE59 to RUN mode-transition time is small, there is no concern with regard to user-perceptible latencies for the simple policy. However, a transition from SLEEP-Display to RUN will most likely be noticed by users. The lazy policy has 100% serious lazy errors.

3) Perfect policy: Assuming we can predict user delays perfectly, we can choose to put the system into the SLEEP-Display mode and wake it up 160ms before the next user input or put the system into the IDLE59 mode. The energy saving can be computed using Equation (3). This is called the $perfect$ policy since it gives the upper bound on energy savings based on the system power model.

4) Prediction policies: We can also use delay prediction, discussed in the previous sections, for DPM/DVS. The energy saving for such prediction policies can be computed using Equation (5). However, there are two concerns with respect to prediction errors. First, in the case of overestimation, the system will wake up from the SLEEP-Display mode upon a user input and enter the RUN mode directly, resulting in

a lazy error, which may be serious. Second, in the case of underestimation, the system wakes up and transfers to the IDLE59 mode after the predicted delay to get ready for the user input and will not be able to fully exploit the idle time to reduce energy by remaining in the SLEEP-Display mode. Therefore, we report the average user delay underestimation error. We refer to the DPM/DVS policies based on user delay prediction by the history-based and psychological models as $history$ and $psychological$, respectively.

Suppose in the GUI shown in Fig. 4, it actually takes a user 1200ms to physically touch a menu item with a stylus. According to the system power model, the Linux kernel DPM policy will consume about 0.53 Joule of system energy. Using the equations from Section VI-B, the simple, perfect and lazy policies will reduce it by about 24%, 38%, and 32%, respectively. The prediction policies will reduce the energy up to 38%, depending on the prediction error. If the prediction error is below 25%, the energy reduction will be between 28% and 38%. In view of the 100% serious lazy errors incurred by the lazy policy, the prediction policies are better.

C. System implementation issues

When a system is put into an energy-saving mode like SLEEP, the processor is stopped, unable to accomplish any regular operating system (OS) maintenance tasks such as *kswapd* and *kupdated* in Linux. There are two solutions to this problem. The first solution is to treat any kernel event as a scheduled user input. The system determines whether and how long to stay in the SLEEP-Display mode based on the next scheduled kernel event and the predicted user delay. This is possible since most Linux kernel events are scheduled with intervals on the order of seconds. The second solution is to delay non-urgent kernel event handling until the next user input or until a certain period of time has elapsed. Another issue is that many passive interactive applications may still perform certain trivial activities when waiting for user input, such as a blinking cursor. Activities with short periods are not compatible with the policies using the SLEEP-Display mode previously addressed. However, activities with reasonably long periods can be handled as a scheduled user input. For example, if the cursor blinks once per second, there is still enough time for the SLEEP-Display mode to save energy. To focus on the investigation of user delay prediction, we ignored these issues in this work by using experiments based on replaying pre-collected usage traces through the system power model.

VIII. Experimental setup

In this section, we describe the benchmark applications, usage traces and power measurement setup used in this study.

A. Benchmarks

An application environment, called Qtopia [2], for Linux-based handheld systems is shipped with the Sharp Zaurus SL-5500 PDA. The source code for its applications is freely available under the General Public License. We consider four applications from Qtopia version 1.5.0 as our benchmarks. They are $Calculator$, $Filebrowser$, Go, and $Solitaire$. $Calculator$ is as described in Section I. $Filebrowser$ is a GUI application for listing files in the local storage. Go and $Solitaire$ are two popular games well known by their names.

Based on an analysis of their source code, STDs were extracted. The STD for $Calculator$ is the same as that shown in Fig. 3. The STD for $Filebrowser$ has five states after simplification: $Menu$ after the user chooses one item from any menu, $ItemClicked$ after the user taps an item in the file list, $Arrow$ after the user taps an arrow on the menu bar (to go back or go upward one level in the directory hierarchy), Dir after the user taps "Dir" on the menu bar (it waits for the user to select an item from the "Dir" menu), and $Sort$ after the user taps "Sort" on the menu bar (it waits for the user to select an item from the "Sort" menu). However, only $ItemClicked$ and $Arrow$ frequently appeared in the usage traces collected. The STDs for both Go and $Solitaire$ only have a single state, which waits for the user's next step: placing a stone or moving a card. The parameter values for user delay models were derived for the perceptual, cognitive and motor processes. The psychological model predictions were used as the initial values for history-based predictions. $Calculator$ requires the least cognitive processing. Therefore, its psychology-based predictions are close to the mean and the standard deviations are relatively small.

B. Usage trace collection

We collected system-user interaction traces for real usage of the benchmarks on the PDA. We inserted $gettimeofday()$ at the beginning and end of the event handlers (usually $slots$ in Qt [32]) in the benchmark applications to record the time by microseconds. This records when the system finishes its response, and when it starts processing an incoming event and changes state. The state name is also recorded. This method counts the delay of the system in generating an event (called $signal$ in Qt) after receiving a user input as part of the user delay instead of the system busy time. Based on our experience, the error thus introduced is negligible.

Two users participated in the collection of traces. One is a male graduate student majoring in Engineering while the other is a female graduate student majoring in Social Sciences. Both are veteran computer users. Before trace collection, they were already familiar with real calculators and file browsing applications. The male user was also familiar with the $Patience$ game in $Solitaire$. The female user was not. Neither of them knew how to play Go. They were taught the rules for playing the games and can be regarded as beginners. They were given instructions on how to operate the PDA and use the benchmark applications. They were also given time to play with the benchmark applications on the PDA to get acquainted. Then they were asked to complete the tasks as described in Table II. Each user was asked to perform the tasks once per day and for a total of three days. The total times of traces for users 1 and 2 were 36.4 and 13.5 minutes, respectively. The traces were collected in an office environment. Disturbance was reasonably minimized. The users were told to perform the tasks in the same way that they would in real life.

TABLE II
TASKS FOR USAGE TRACE COLLECTION

Benchmark	Task
Calculator	Compute formula, each with three two-digit numbers and two operations
Filebrowser	Find a given file on the local storage
Go	Play a new game for several minutes
Solitaire	Play a new Patience game for several minutes

C. Trace replay

We replayed the collected traces through the system power model detailed in Section VII-A. We applied the power management policies described in Section VII-B in different replays to gauge their performance. Since the traces were pre-collected, we implicitly assumed that user behavior remained unchanged after the users encountered perceptible latencies. Therefore, the energy savings obtained by policies that incur user-perceptible latencies are only ideal, and are likely to be discounted in reality due to productivity degradation. However, since the relationship between productivity degradation and user-perceptible latencies is different in different applications, such an assumption makes our evaluation more general. Moreover, since we evaluate prediction methods using tradeoffs between such ideal energy savings and user-perceptible latencies, we believe that there is enough information for system designers to make these tradeoffs based on the nature of human-computer interaction in their applications.

IX. EXPERIMENTAL RESULTS

In this section, we present experimental results using the setup described in the previous section.

A. Energy savings

The system energy consumption is obtained by running the usage traces through the system power model with different DPM/DVS policies. In Fig. 8, we give energy savings against the baseline (Linux policy). These include the simple, lazy, perfect, and psychology/history-based prediction policies. In the history-based prediction policy, the pessimism factor β is assumed to be 0.4 and the last seven delay records are used starting with the psychological prediction. The percentage of lazy error (PLE) (the percentage of all predictions that are overestimations) and PSLE are reported in Fig. 9 for the psychological (solid lines) and history-based (dashed lines) policies. We adopt $50ms$ as the human perceptual threshold. Notably, the lazy policy's energy savings are closest to the perfect policy's in three of the four benchmarks. Only in $Calculator$, it is less energy-efficient than the other policies due to the fact that user delays for $Calculator$ tend to be much shorter and predictable (see Section VI-B). However, the lazy policy has 100% serious lazy errors. Therefore, it will be useful only when system latencies are tolerable and the user delays are relatively long.

It is worth noting that the psychological policy performs better for $Calculator$ and $Filebrowser$ than Go and $Solitaire$ compared to the history-based policy. This is due to the fact that operating $Calculator$ and $Filebrowser$ is cognitively much simpler and their cognitive processes are better modeled by the Hick-Hyman Law.

B. Tradeoff between lazy errors and energy savings

For the psychological policy, the lazy errors are very few since this model is conservative. However, for the history-based policy, the pessimism factor controls the tradeoff between the lazy errors and energy savings. Figs. 10 and 11 show how the energy saving changes with the percentage of lazy errors for $Calculator$ and $Filebrowser$ as the pessimism factor varies from 0.2 to 1.3 for the history-based policy. It also shows the tradeoff point for the psychological policy. To achieve the same energy saving as the psychological policy, the history-based policy needs to make a lot more lazy errors. This demonstrates the superiority of the psychological policy when avoiding lazy errors is important. These figures show that psychology-based prediction works better for $Calculator$ than $Filebrowser$ compared to history-based prediction since operating the former is cognitively much simpler.

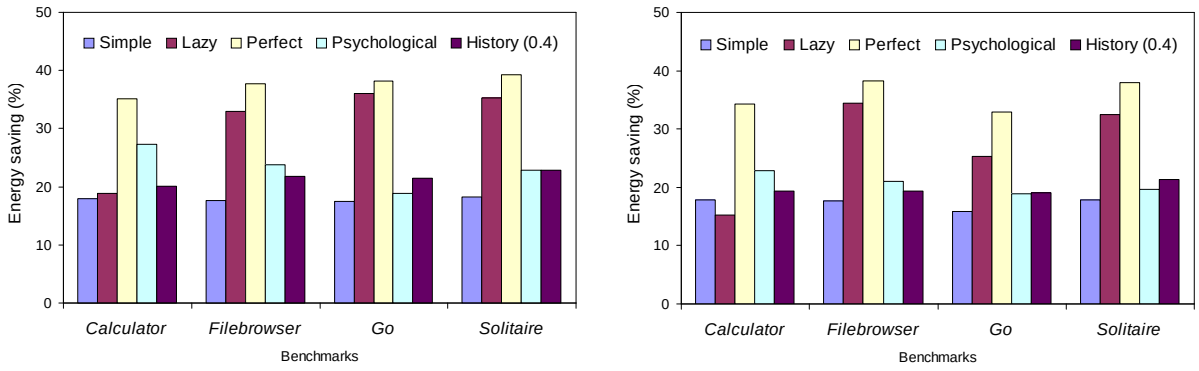
C. State-awareness

To show the benefit of being state-aware in history-based prediction, Figs. 10 and 11 also show the tradeoff curves for state-unaware history-based prediction, which uses the last seven observed delays of the application to predict the next delay for the same application without knowing its states. It is application-based instead of state-based. It is clear that state-based prediction is better in general. Moreover, the STDs of $Calculator$ and $Filebrowser$ are relatively simple. We expect the advantage of state-awareness to be larger for more complicated applications with a larger number of STD states.

We also note that state-awareness does not benefit user 2 for $Calculator$, meaning the user did not behave very differently when responding to the two STD states. While the mean user delays for the $Calculator$ states in user 1's trace differ a lot, those in user 2's trace do not. This illustrates the difference in behavior between different individuals when interacting with a computer.

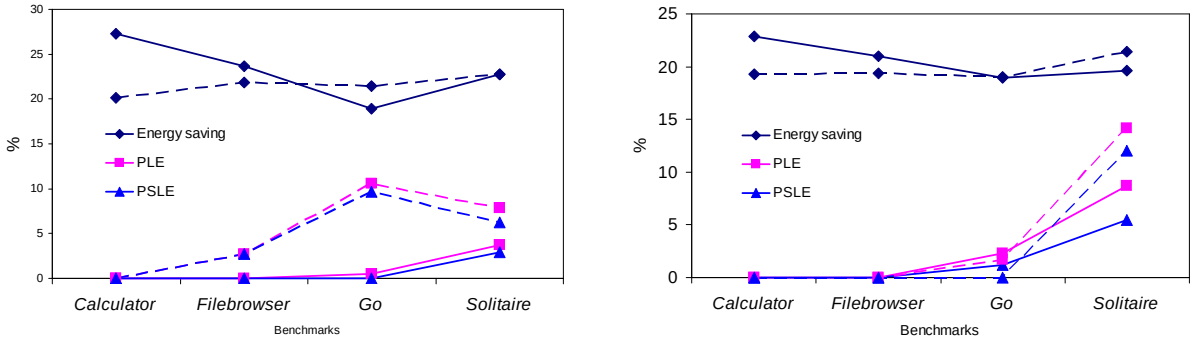
D. Power-saving mode: Power, transition time and policy

It is obvious from the theoretical analysis in Section VI-B that the energy saving will be more when the power of the power-saving mode decreases, given that all other aspects remain unchanged. However, as shown by the system model in Fig. 7, a power-saving mode with lower power usually incurs a longer transition time back to the RUN mode. A longer transition not only increases energy overhead but may also cause a user-perceptible latency if it is larger than the human perceptual threshold, 50ms. First, we would like to investigate how the power-saving mode's power and transition time will impact energy savings. Fig. 12 plots the curves of energy saving vs. the SLEEP-Display mode power for



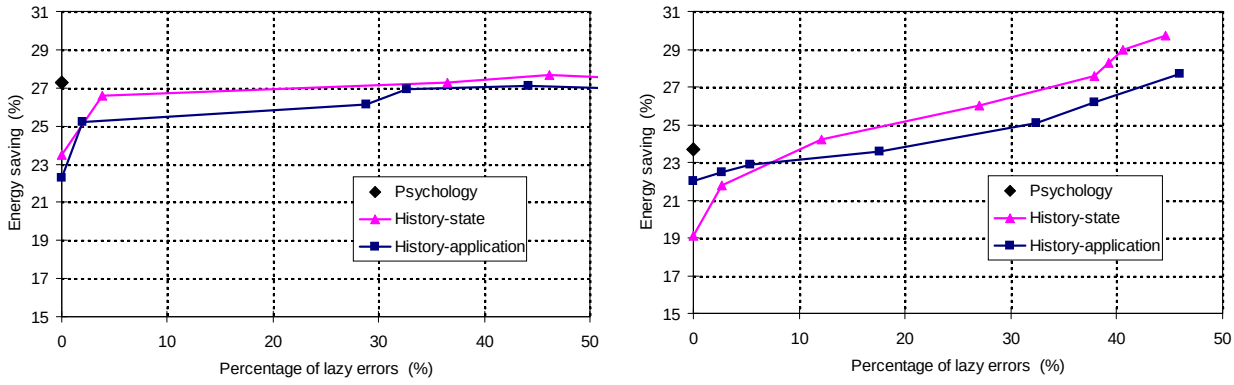
(a) User 1 (b) User 2

Fig. 8. Energy savings based on predicted user delays



(a) User 1 (b) User 2

Fig. 9. Percentage of lazy errors and energy savings



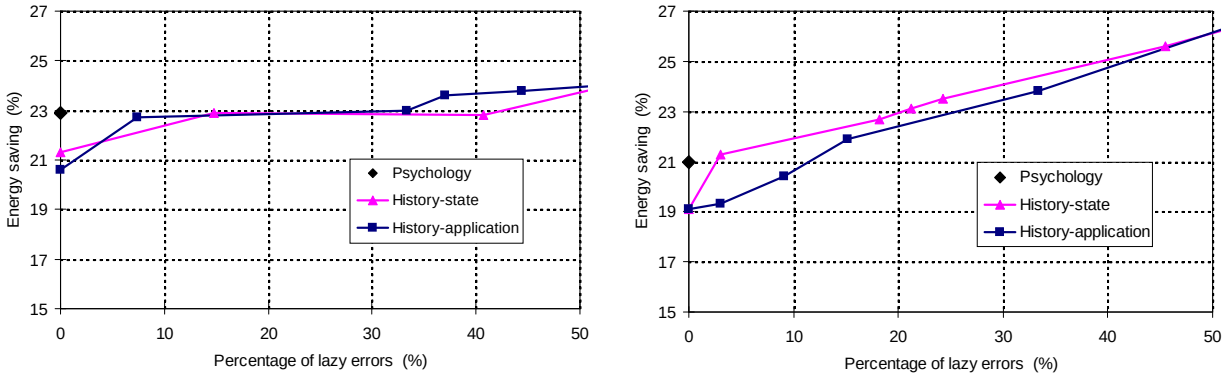
(a) Calculator (b) Filebrowser

Fig. 10. Tradeoff between percentage of lazy errors and energy savings for user 1.

different policies (Perfect, Lazy, and History) and two different transition times (160ms and 40ms), all for user 1.

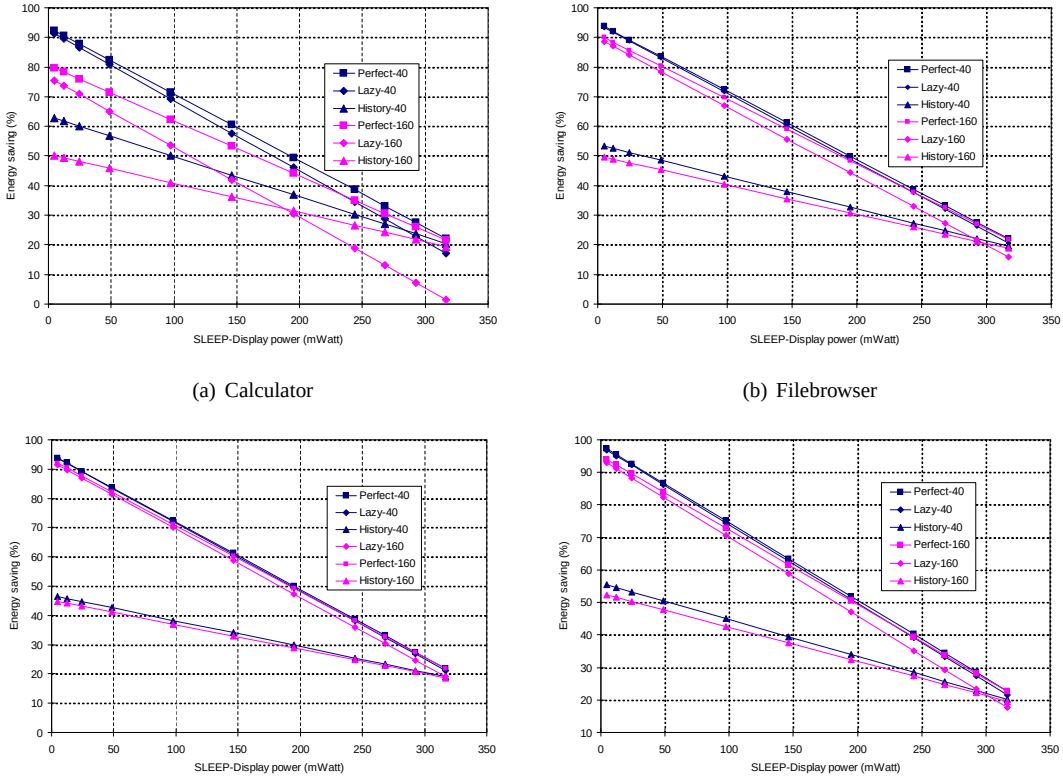
For all policies, energy savings increase linearly as the SLEEP-Display mode power decreases for all benchmarks. For $Calculator$, the user delays are relative short, which leads to relatively large energy saving increase when the transition time decreases from 160ms to 40ms. For other benchmarks, such a decrease does not bring much energy benefit. This is because $Calculator$ has relatively short user delays. Therefore, decreasing the transition time below the human perceptual threshold is worthwhile only when the expected user delays are short. On the other hand, decreasing power-saving mode's power is always effective. For example, if there are two power saving modes with power/transition time being 150mW/45ms and 200mW/1ms, respectively, the first one will save much more energy than the second for most interactive applications without causing serious lazy errors.

The lazy policy's advantage in energy saving over the



(a) Calculator (b) Filebrowser

Fig. 11. Tradeoff between percentage of lazy errors and energy savings for user 2.



(a) Calculator (b) Filebrowser

(c) Go (d) Solitaire

Fig. 12. Energy saving changes with mode power and transition time

history-based policy increases as the power decreases. When the transition time is smaller than the human-perceptible threshold of 50ms, the lazy policy can be safely used without causing user-perceptible latencies.

X. Discussions

In the previous section, we showed the effectiveness of combining DPM/DVS with user delay prediction. We discuss several related issues next.

A. STD extraction

As may be apparent to the reader, how an STD is extracted for an interactive application is important for user delay prediction. For example, if *Calculator* is modeled with only one state, which may seem natural, the user delay prediction will degrade as demonstrated for the history-based model for user 1. Also, the not-so-good performance of user delay models for *Go* and *Solitaire* can be partially attributed to modeling them with a single-state STD. Adding states into the STD has the potential to improve intra-state consistency and the tradeoff between energy savings and lazy errors.

B. User delay uncertainty

Since an STD state presents relatively consistent information to the user and requires relatively consistent user reaction, we expect the corresponding user delays to be more predictable. However, there are two factors that contribute to the uncertainty in user delays, as discussed next.

First, the presented information may vary for different occurrences of the same state. For example, it can be dynamic in the case of file browsers and text readers. Such intra-state inconsistency can be reduced by using more states. The dynamic information problem can be partially solved by obtaining information about the content, such as obtaining the number of items in the current directory and the size of the text to be shown. The GUI toolkit-based implementation takes care of this readily.

Second, a state may require complicated user reactions, or permit very different user reactions. For a complicated state, the user may respond only after reading different subset of the information presented in a state. For example, a $Solitaire$ player may not examine all stacks of cards before responding. In other words, such a state represents multiple subtasks to a user. However, the uncertainty is alleviated by the fact that there is usually one subtask most often performed by the user. For example, a $Filebrowser$ user most likely will read the file list and click a file from the list; a $Word$ user will type letters most of the time. Therefore, the most frequently-performed subtask(s) can be used to build the psychological model, as we did for Go and $Solitaire$. The history-based prediction policy automatically benefits from this fact.

C. Improving prediction accuracy

We showed in Section VI-B that the energy saving ratio is directly related to the user delay prediction accuracy. The experimental results also demonstrated that more energy can be saved when delay can be predicted more accurately. To better utilize user delays for energy reduction, improving prediction accuracy will be important.

In this work, we used the psychological models and values of their coefficients from previous work, which were not based on PDAs. Since our evaluation is based on a PDA, this modeling discrepancy may prevent it from benefiting more from our methodology. We expect psychological experiments on humans using PDAs to generate more PDA-specific user delay models. Moreover, as we have pointed out, the cognitive delay models based on the Hick-Hyman Law may contribute most to prediction errors for the psychological model. Better cognitive delay models are likely to increase energy savings. Moreover, users may differ from each other in their perceptual, cognitive, and motor delays. The same user may improve his/her usage skills due to the learning process. The environment or context may also change user behavior. Although the history-based model automatically adapts, the psychological model does not. It would be interesting to see how adaptation of user delays to changing human behavior would improve prediction accuracy and increase energy savings.

D. Hardware support for interactive systems

Since the display has to be always on during user-computer interaction, changing the operation mode of other components of the system should be independent of the display. For example, the system should be able to put its processor to SLEEP without affecting its display. To the best of our knowledge, there is no such hardware support in commercially-available mobile computing systems. For displays that need refreshing, the frame-buffer and the control unit transferring data from it to the display must be on too. In most systems, these components can be on with others power-managed. For example, in the StrongARM/PXA family systems-on-chip, the frame-buffer sits in the main memory and the LCD controller is one of the peripheral units. They can be active while many other units are power-managed [33].

Another solution is to use a display that maintains its content itself. In the μSleep work [21], an LCD with an integrated frame-buffer was used. When the system was put into the SLEEP mode, the LCD displayed the image in its own frame-buffer automatically without support from the SA-1100 LCD controller. Moreover, bistable LCDs [36]–[38] can maintain the screen content even without the power supply. However, they usually take more time and energy overhead for screen changes. They will be ideal for passive interactive systems that have relatively long user delays.

E. Other ways to benefit from user delay prediction

We showed how user delay prediction can be utilized for DPM/DVS in the scenario in which there is only one application under consideration. In other scenarios, an interactive application may be used with other applications running in the background. A user may enjoy music from the MP3 player, while downloading another MP3 file and playing $Solitaire$ at the same time. Therefore, it would be better if $Solitaire$ notifies the OS about predicted user delays and the OS globally schedules all the processes. The OS scheduler can make use of the predicted user delays to allocate time slots and performance levels to different processes in an energy-efficient fashion.

XI. Conclusions

Based on an analysis of its power characteristics, we found an interactive system spends most of its time and energy waiting for user responses. Therefore, the most effective way to reduce system energy consumption is to reduce the energy consumed during user delays.

We proposed an application-based user delay prediction framework. In this framework, an STD is first obtained for the application to model the interaction between the system and user. User delays are then predicted based on different STD states. Within this framework, two prediction models were proposed. We theoretically showed how prediction errors are related to energy savings for DPM/DVS. The history-based model predicts the actual delay based on recent observations. Since it may overestimate the delay, it should only be used when performance level transition time is not a large concern. In our experiments, it resulted in an average system energy

saving of 20.7% with a relatively small percentage of serious lazy errors. We also showed that exploiting STD states yielded a better tradeoff between lazy errors and energy savings. The psychological model exploits the user interface information further and predicts the lower bound on user delays, $i.e.$, how long it takes the user to read, decide, and move. Our experiments show that an average of 21.9% system energy savings can be obtained with negligible serious lazy errors. We showed that the tradeoff between lazy errors and energy savings achieved by the psychological model is beyond the capacity of the history-based model.

We also showed how DVS can be simply combined with user delay prediction. An average of 17.6% system energy reduction can be easily achieved without introducing any user-perceptible delays. For applications more tolerant of system delays and with longer user delays, an average of 28.9% system energy reduction can be achieved.

ACKNOWLEDGMENTS

The authors would like to thank Yuanyuan Zhang and Pallav Gupta for participating in usage trace collection. They would also like to thank the anonymous reviewers, whose comments significantly improved the paper.

REFERENCES

[1] L. Zhong and N. K. Jha, "Dynamic power optimization for interactive systems," in *Proc. Int. Conf. VLSI Design*, Jan. 2004, pp. 1041–1047.
[2] Qtopia, *http://www.trolltech.com/products/qtopia/*.
[3] K. Flautner, R. Uhlig, S. Reinhardt, and T. Mudge, "Thread-level parallelism and interactive performance of desktop applications," in *Proc. Int. Conf. Architecture Support for Programming Language & Operating Systems*, Aug. 2000, pp. 129–138.
[4] L. Benini, A. Bogliolo, and G. De Micheli, "A survey of design techniques for system-level dynamic power management," *IEEE Trans. VLSI Systems*, vol. 8, no. 3, pp. 299–316, June 2000.
[5] N. K. Jha, "Low power system scheduling and synthesis," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 2001, pp. 259–263.
[6] M. B. Srivastava, A. P. Chandrakasan, and R. W. Brodersen, "Predictive system shutdown and other architectural techniques for energy efficient programmable computation," *IEEE Trans. VLSI Systems*, vol. 4, no. 1, pp. 42–55, Jan. 1996.
[7] C.-H. Hwang and A. C.-H. Wu, "A predictive system shutdown method for energy saving of event-driven computation," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 5, no. 2, pp. 226–241, April 2000.
[8] Q. Qiu and M. Pedram, "Dynamic power management based on continuous-time Markov decision processes," in *Proc. Design Automation Conf.*, June 1999, pp. 555–561.
[9] S. Irani, S. Shukla, and R. Gupta, "Online strategies for dynamic power management in systems with multiple power-saving states," *ACM Trans. Embedded Computing Syst.*, vol. 2, no. 3, pp. 325–346, July 2003.
[10] T. Pering, T. Burd, and R. Brodersen, "The simulation and evaluation of dynamic voltage scaling algorithms," in *Proc. Int. Symp. Low Power Electronics & Design*, Aug. 1998, pp. 76–81.
[11] D. Grunwald, P. Levis, K. I. Farkas, C. B. Morrey III, and M. Neufeld, "Policies for dynamic clock scheduling," in *Proc. Symp. Operating Systems Design & Implementation*, Oct. 2000, pp. 73–86.
[12] T. Šimunić, L. Benini, and G. De Micheli, "Event-driven power management of portable systems," in *Proc. Int. Symp. System Synthesis*, Nov. 1999, pp. 18–23.
[13] J. Pouwelse, K. Langendoen, and H. Sips, "Energy priority scheduling for variable voltage processors," in *Proc. Int. Symp. Low Power Electronics & Design*, Aug. 2001, pp. 28–33.
[14] T. Heath, E. Pinheiro, J. Hom, U. Kremer, and R. Bianchini, "Application transformations for energy and performance-aware device management," in *Proc. Int. Conf. Parallel Architecture & Compilation Techniques*, Sept. 2002, pp. 121–131.
[15] M. Anand, E. B. Nightingale, and J. Flinn, "Ghosts in the machine: Interfaces for better power management," in *Proc. Int. Conf. Mobile Systems, Applications, & Services*, June 2004, pp. 23–35.
[16] K. Flautner, S. Reinhardt, and T. Mudge, "Automatic performance setting for dynamic voltage scaling," in *Proc. Ann. Int. Conf. Mobile Computing & Networking*, July 2001, pp. 260–271.
[17] J. R. Lorch and A. J. Smith, "Improving dynamic voltage scaling algorithms with PACE," in *Proc. ACM SIGMETRICS*, June 2001, pp. 50–61.
[18] ——, "Using user interface event information in dynamic voltage scaling algorithms," in *Proc. Int. Symp. Modeling, Analysis & Simulation of Computer & Telecommunications Systems*, Oct. 2003, pp. 46–55.
[19] L. Zhong and N. K. Jha, "Graphical user interface energy characterization for handheld computers," in *Proc. Int. Conf. Compilers, Architecture, and Synthesis for Embedded Systems*, Nov. 2003, pp. 232–242.
[20] N. Kamijoh, T. Inoue, C. M. Olsen, M. T. Raghunath, and C. Narayanaswami, "Energy trade-offs in the IBM wristwatch computer," in *Proc. Int. Symp. Wearable Computers*, Oct. 2001, pp. 133–140.
[21] L. S. Brakmo, D. A. Wallach, and M. A. Viredaz, "μSleep: A technique for reducing energy consumption in handheld devices," in *Proc. Int. Conf. Mobile Systems, Applications, & Services*, June 2004, pp. 12–22.
[22] A. Wasserman, "Extending state transition diagrams for the specification of human-computer interaction," *IEEE Trans. Software Engineering*, vol. 11, no. 8, pp. 699–713, Aug. 1985.
[23] P. Brockwell and R. A. Davis, *Introduction to Time Series and Forecasting*. New York: Springer-Verlag, 1996.
[24] B. Schneiderman, "Response time and display rate in human performance with computers," *ACM Comput. Surv.*, vol. 16, no. 3, pp. 265–285, Sept. 1984.
[25] T. Goodman and R. Spence, "The effect of system response time on interactive computer aided problem solving," in *Proc. 5th Annual Conf. Computer Graphics & Interactive Techniques*, Aug. 1978, pp. 100–104.
[26] S. K. Card, T. P. Moran, and A. Newell, *The Psychology of Human-Computer Interaction*. Hillsdale, NJ: Lawrence Erlbaum Assoc., 1983.
[27] R. P. Carver, *Reading Rate: A Review of Research and Theory*. San Diego, CA: Academic Press, Inc., 1990.
[28] W. E. Hick, "On the rate of gain of information," *Quarterly J. Experimental Psychology*, no. 4, pp. 11–36, 1952.
[29] R. Hyman, "Stimulus information as a determinant of reaction time," *J. Experimental Psychology*, no. 45, pp. 188–196, 1953.
[30] P. M. Fitts, "The information capacity of human motor system in controlling the amplitude of movement," *J. Experimental Psychology*, no. 47, pp. 381–391, 1954.
[31] I. S. MacKenzie and W. Buxton, "Extending Fitts' law to two-dimensional tasks," in *Proc. Conf. Human Factors in Computing Systems*, 1992, pp. 219–226.
[32] Qt, *http://www.trolltech.com/*.
[33] Intel PCA processors manuals and guides, *http://www.intel.com/design/pca/applicationsprocessors/manuals/index.htm*.
[34] M. A. Viredaz and D. A. Wallach, "Power evaluation of a handheld computer," *IEEE Micro*, vol. 23, no. 1, pp. 66–74, Jan./Feb. 2003.
[35] National Instruments, *http://www.ni.com/*.
[36] Kent Displays, Inc., *http://www.kentdisplays.com/product/modules.htm*.
[37] ZBD Displays Ltd., *http://www.zbddisplays.com*.
[38] Nemoptic Advanced LCD Technologies, *http://www.nemoptic.com*.