

Combining Compiler and Operating System Support for Energy Efficient I/O on Embedded Platforms

Ripal Nathuji
School of Electrical and
Computer Engineering
Georgia Institute of
Technology
Atlanta, GA 30032
rnathuji@ece.gatech.edu

Balasubramanian
Seshasayee
College of Computing
Georgia Institute of
Technology
Atlanta, GA 30032
bala@cc.gatech.edu

Karsten Schwan
College of Computing
Georgia Institute of
Technology
Atlanta, GA 30032
schwan@cc.gatech.edu

ABSTRACT

Mobile and embedded platforms have experienced dramatic advances in capabilities, largely due to the development of associated peripheral devices for storage and communication. The incorporation of these I/O devices has increased the overall power envelope of these platforms. In fact, system-level power consumption of mobile platforms is often dominated by peripheral devices. Since battery technologies alone have been unable to provide the lifetimes required by many platforms, in order to conserve energy, most devices provide the ability to transition into low power states during idle periods. The resulting energy savings are heavily dependent upon the lengths and number of idle periods experienced by a device. This paper presents an infrastructure designed to take advantage of device low power states by increasing the burstiness of device accesses and idle periods to provide a reduced power profile, and thereby an improvement in battery life. Our approach combines compiler-based source modifications with operating system support to implement a dynamic solution for enhanced energy consumption. We evaluate our infrastructure on an XScale-based embedded platform with a Linux implementation.

1. INTRODUCTION

The portability and availability of mobile devices have made them an attractive platform for both personal and commercial applications. Commonly, the determining factor of whether or not to utilize these platforms is the battery life the end user can expect to experience. Although the addition of peripheral I/O devices to mobile platforms has allowed for increased capabilities, the resulting increase in power consumption has made effective energy management a critical design element of embedded platforms. Many of the methods that may be used to intelligently manage system energy, such as spinning down disk drives or shut-

ting down communication devices, require dynamic software techniques integrated into the operating system and application code in addition to hardware optimizations [5, 9]. This paper follows previous work in developing a software based infrastructure for energy management [4, 8].

As opposed to hardware optimizations implemented at manufacturing time, it is difficult to predict the reduction in productivity that can result during execution time from dynamic techniques for reducing energy consumption. Any dynamic approach must therefore be careful not to unduly affect system performance. In the case of reducing the energy consumption of peripheral devices, one approach is to increase the burstiness of device usage periods to improve the extents of idle times. Any approach that utilizes this principle, however, must not substantially reduce user-centric measures of I/O service like throughput or response time.

When designing software-based power management schemes, design alternatives include whether to incorporate additional or modified programming APIs, utilize compiler driven hints, or devise a solution completely internal to the operating system. Solutions that require adherence to a new interface for performing device I/O provide increased information to the power management subsystem at the cost of transparency and backwards compatibility [18]. Our approach provides a solution that requires no modifications to existing APIs and is completely transparent to the programmer. Moreover, while we utilize source annotations in our algorithms and infrastructure to provide improved performance, these compiler-driven hints are not required, thereby allowing us to also support legacy binaries.

The approach to power management taken in our work involves collaboration between the compiler and the operating system. In particular, our solution combines runtime device usage information from monitoring state with compiler annotation-based hints that correlate device usage to application processes. This information is abstracted into a set of operating system data structures termed device windows and provided to the process scheduler. The resulting system, termed the Energy Efficient Scheduling Infrastructure (EESI), attempts to order processes so as to (1) increase the burstiness of device accesses, and (2) provide increased idle periods to exploit low power states of peripheral devices

that utilize timeout-based sleep mechanisms.

Our approach is implemented in the Linux operating system. This implementation includes kernel modifications to incorporate new data structures, run-time monitoring, support for annotations, and the resulting process scheduler. The system is evaluated on a representative portable device, namely an experimental platform based upon the Intel XScale processor. Power measurements of typical application scenarios result in power savings of up to 18% and up to 17% improvement in time spent in low power modes.

2. RELATED WORK

Approaches to reducing the energy consumption of portable devices generally focus on various avenues for optimization. Recent processor architectures, such as the Intel XScale, provide support for dynamic frequency and voltage scaling (DFS/DVS). Since the dynamic power consumption of a CPU is proportional to the product of frequency and voltage squared [6], these techniques can be effective in reducing power consumption during program execution [17]. Adhering to application-level requirements such as task deadlines while performing DFS/DVS requires adjustments to process schedules as explained in [16]. A design framework for exploring power/performance trade-offs when developing hard real-time systems is proposed in [3]. The ability to obtain energy savings by performing application-level adaptations for multi-media applications is investigated in [11]. Other methods for utilizing frequency scaling include a compiler/OS collaborative system for frequency scaling [1].

An accounting-based approach to provide system lifetime guarantees is presented in [20]. By making power a first class system resource, the approach strictly allocates power to processes, and is then extended with modified scheduling algorithms in [19]. While providing lifetime guarantees, this approach does not provide any performance or user-level guarantees such as task deadlines or throughput. In [2], the author proposes energy-aware scheduling techniques using energy accounting based upon hardware performance counters.

A substantial amount of the research for utilizing device idle periods is concerned with communication devices. The benefits of a separate low power channel to determine when to turn off these devices is investigated in [13]. In [5], the authors modify the 802.11 protocol at the client and base station to develop a collaborative approach to putting a wireless device to sleep. By developing a new I/O interface, a mechanism for energy-aware resource usage is described in [18].

When considering multiple devices, the problem of putting devices to sleep becomes more complicated. Given a predetermined task schedule and device usage list, [15] presents an algorithm to determine a schedule for device sleep/active states. Given perfect knowledge of device usage information, an algorithm that schedules tasks so as to maximize device idle periods is given in [7].

In [14], the authors present a multi-processor scheduling algorithm that utilizes processor-cache affinity information to schedule tasks so as to minimize cache overheads. We attempt to build a power analogue of this approach where

affinity information is contained in our device windows data structures. We improve upon a completely dynamic approach by incorporating additional information provided by a source annotation scheme. This integration includes the creation of algorithms to trade off the benefits of both dynamic state and statically placed compiler hints. The resulting solution intelligently combines both of these inputs to provide a power-enhanced process scheduler.

3. MOTIVATION

3.1 Aggregating Device Accesses

When using timeout-based mechanisms to place devices into low power modes, both the sizes and distribution of idle periods experienced by the device determine the amount of energy that is saved. Given a particular idle period, a device is able to transition into a low power state only if the idle period is longer than the timeout period assigned to the device. Moreover, if a device is able to transition into a low power state, the amount of time actually spent conserving energy (t_{sleep}) is equal to the length of the idle period (t_{idle}) minus the sum of the timeout period (t_{to}), the time to perform an idle-to-sleep state transition (δ_{hl}), and the time to perform a sleep-to-idle transition (δ_{lh}) for the subsequent request.¹ This relationship is provided by Equation 1. Since t_{to} , δ_{hl} , and δ_{lh} are all constant terms, this tells us that to maximize the effectiveness of a sleep period, the idle time should be maximized so as to amortize the cost of timeouts and state transitions in the most effective manner.

$$t_{sleep} = t_{idle} - (t_{to} + \delta_{hl} + \delta_{lh}) \quad (1)$$

In addition to the sizes of idle periods, their distributions also play a key role. Figure 1 illustrates an example of three periods of device activity. In Figure 1(a), short idle periods are distributed amongst the requests, ending with a final idle period leading to a state transition. Figure 1(b) depicts the same scenario, but with the three periods of device activity aggregated together. Though the active time of the device remains the same, it is now able to transition to sleep mode sooner. Assuming the subsequent request that “awakens” the device arrives at the same point in time for both scenarios, the bursted approach maximizes Equation 1 for the final idle period by increasing t_{idle} . Moreover, if there is a subsequent request immediately prior to the point where the device times out in the non-bursted scenario, the device experiences no sleep time at all, whereas in the bursted scenario some savings are still achieved. Therefore, bursting device accesses not only provides longer idle periods resulting in increased sleep time, but can also create periods of low power state that would otherwise not occur.

To fully illustrate the energy benefits of bursting device accesses, we consider the energy consumption of a device over a given time period given by Equation 2. Regardless of whether or not accesses are bursted, the active time is equivalent. Therefore, the first component of the equation can be ignored when comparing the energy footprint of the

¹It should be noted that in our notation t_{sleep} , δ_{hl} , and δ_{lh} are implicitly subcomponents of t_{idle}

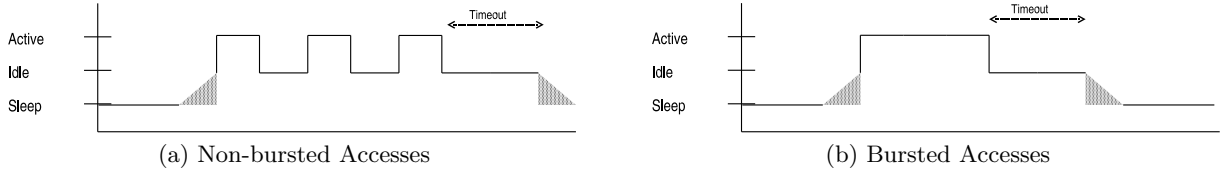


Figure 1: Device State with Varying Access Patterns

approaches. In other words, since $(t = t_{active} + t_{idle})$, any benefit derived from bursting accesses is dependent upon how much of t_{idle} , if any, is distributed into t_{sleep} , $n_{hl}\delta_{hl}$, and $n_{lh}\delta_{lh}$, where n_{hl}/n_{lh} denote the number of state transitions. It has already been shown that access bursting increases t_{sleep} . Another, possibly significant, implication of access bursting is to reduce n_{hl}/n_{lh} . The energy benefits of this effect are illustrated by Equation 2.

$$E(t) = P_{active}t_{active} + P_{idle}(t_{idle} - t_{sleep} - n_{hl}\delta_{hl} - n_{lh}\delta_{lh}) + P_{sleep}t_{sleep} + P_{hl}n_{hl}\delta_{hl} + P_{lh}n_{lh}\delta_{lh} \quad (2)$$

3.2 A Process Scheduling Approach

Having illustrated and discussed the benefits of bursting devices accesses, a question that arises is where in the system access aggregation should be performed. One option is to provide device bursting at the lowest possible layer, the device driver [12]. The problems of such an approach are twofold. First, bursting accesses at the driver may alter the delay assumptions made by higher level protocols. For example, bursting network packets at the driver level sometimes requires the delaying of packets. This may cause TCP to estimate varying round-trip-times, which in turn can result in changes to throughput for the connection. A second drawback to performing bursting at a low level is that it becomes difficult to cleanly integrate bursting with other power management techniques such as DFS/DVS. Instead, it would be beneficial to integrate various power management techniques at a higher level in the operating system. Therefore, in this paper, we perform device access bursting using the process scheduler.

3.2.1 Scheduling for Access Bursting

The operating system process scheduler can be utilized to perform access bursting at a coarse level. Let us assume that the scheduler is aware of the device(s) a given process will access during its next execution phase. When the current process uses up its time quantum (or relinquishes control), the scheduler attempts to choose a process that requires a similar set of devices, thereby bursting accesses to used devices and extending the idle periods of unused devices. Figure 2 illustrates an example scenario with two devices. Processes one and three access device 1, and process two accesses device 2. By comparing Figure 2(b) to Figure 2(a), we observe that by simply reordering the FIFO scheduling order to a bursted approach, the maximum idle periods experienced by both devices at least double in length. It should also be emphasized that these idle time benefits are obtained while still providing the same amount of service (*i.e.* execution time) to all processes. Indeed, processes are only “reshuf-

fled” within the bounds of a scheduling epoch.

Implementing a scheduler that behaves as the one in Figure 2 raises a few issues. First, there is the issue of how the scheduler determines what type of device set would be optimal for the next process. This set would likely be determined using some sort of system state such as recent device activity behavior. Second, there must be some type of likelihood indicator to determine what devices a process will use when scheduled. For the moment, let us assume that given a process p , the maximizing function $M(p)$ provides a heuristic of how suited a process is to continue bursting accesses on devices and/or extend idle periods. Given $M(p)$, Algorithm 1 provides the EESI scheduling algorithm which attempts to provide bursty access behavior.

ALGORITHM 1.

\forall Processes $i, j \in \text{Runnable}$
Choose Process i s.t. $M(i) \geq M(j)$ ($\forall j \neq i$)

3.2.2 Device Windows - Data Structures for Device Usage Correlation

A remaining issue that needs to be addressed is how the scheduler obtains system-level device usage patterns and the methodology for predicting future accesses by a process. In our system, we utilize special kernel-level data structures for this purpose called device windows. There are two types of device windows, *system device windows* and *process device windows*. System device windows are used to collect data for determining the activity level of devices from a system perspective. Therefore, there is an individual system device window for each device. Similarly, there are process device windows associated for each task that are used to predict whether that device will be used the next time the process executes on the CPU.

Semantically, device windows are used in a very simple manner. For each type of device, there is an associated system device window value threshold and process device window value threshold. Given these values, if the value of a system device window is greater than its threshold, that device is considered active in the system, and the scheduler will attempt to burst accesses to it. Similarly, if the value of a process device window is greater than its threshold, the scheduler will predict that the process will access the device whenever it next executes on the CPU. Using the notions of values and thresholds, Table 1 outlines the parameters that allow us to define the maximizing function $M(p)$ for our EESI scheduler in Equation 3. A more detailed description of device windows can be found in [10].

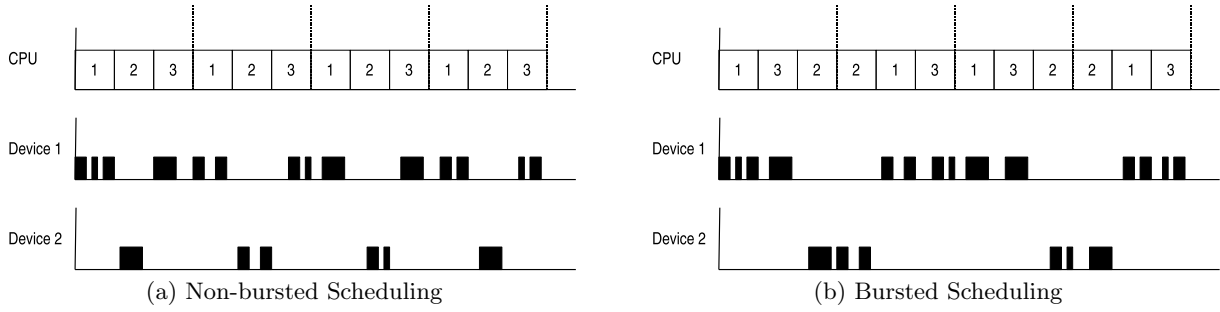


Figure 2: Access Distributions with Varying Scheduling Approaches

Table 1: Device Window Parameter Definitions

Parameter	Definition
$S_{0,i}$	1 if System device window value less than threshold for device i , 0 otherwise
$S_{1,i}$	1 if System device window value greater than threshold for device i , 0 otherwise
$P_{0,i,p}$	1 if Process device window value less than threshold for process p and device i , 0 otherwise
$P_{1,i,p}$	1 if Process device window value greater than threshold for process p and device i , 0 otherwise
$W_{i,p}$	Process window device value for process p and device i
$W_{i,max}$	Maximum window device value for device i
λ_i	Weight value for device i

$$M(p) = \sum_{i=1}^K \lambda_i (S_{1,i} P_{1,i,p} W_{i,p} + S_{0,i} P_{0,i,p} (W_{i,max} - W_{i,p})) \quad (3)$$

4. SYSTEM DESIGN

Section 3 has presented the necessary motivation and background information for our approach. This section describes the design of our solution and the trade-offs it considers. An aspect of our infrastructure that needs to be defined is the mechanism used to set and update window values. In this paper, we investigate the use of dynamic usage state combined with device usage hints placed into application source code by an annotator as a device windows update mechanism.

4.1 Dynamic State vs. Static Hints

To begin discussion, we consider the pros and cons of a reactive approach based upon dynamic state vs. a proactive approach based upon source annotation. In order for our scheduler to function correctly, it must be able to predict whether a process will use a device with some degree of certainty. In a completely reactive approach, it is difficult for the system to predict when a process will access a device after prolonged idle periods. In other words, since any reactive approach is based upon recent history information, the system is unable to predict when a process that has not used a device within a history period will begin to do so again. This is a distinct drawback of a reactive approach. Moreover, a dynamic approach implicitly assumes that once activity is detected, it is an indicator of subsequent activity. This can lead to inaccurate predictions for applications that have isolated bursts of device activity. These two issues do not exist with a system driven by static compiler hints.

Static hints can aid in isolating portions of application code that access a device by marking portions of code that access relevant operating system or library calls. Unfortunately, for many devices a system call that appears to access a device at annotation time may not consistently do so during execution. This phenomenon may occur due to a variety of system interactions including buffering or prefetching, effects that are common with high latency devices such as disks. Therefore, a pure annotation based scheme cannot provide an ideal solution on its own. In Sections 4.2 and 4.3 we illustrate how both reactive and proactive approaches can be utilized with the device windows abstraction. We then discuss a collaborative approach that utilizes both of these design elements and is the basis of the EESI system in Section 4.4.

4.2 Dynamic Device Usage Information

A dynamic scheme is dependent upon the creation and use of history information to perform predictions of device usage behavior. Figure 3 illustrates how device windows are incorporated into the scheduling mechanics of a reactive system. Whenever a process is removed from the CPU, run-time monitoring state of device usage is used to update system device windows and the device windows of the particular process. To describe the manner in which these windows are updated, we first present how the abstraction of device windows is implemented.

Both system and process device windows are treated as statically sized sliding windows. Each time a process is removed from the processor, each of its device windows is shifted to the right. For all devices that were used during the execution period of the process, the ‘last’ (leftmost) bit is set in the respective process device window. The system device windows are updated in a similar fashion each time a process is removed from the processor.

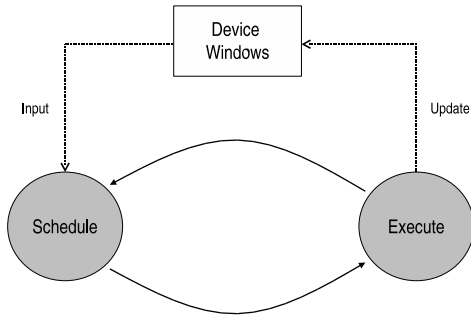


Figure 3: Dynamically Updating Device Windows

Given this update mechanism, the value of device windows can be interpreted in one of two ways. Device windows can be compared by the binary values of their bits. This approach exponentially reduces the significance of a bit whenever the window is shifted. We use this method of calculating values for process device windows. For this type of window the maximum value is $(2^l - 1)$. Whenever the age of a particular bit is not significantly relevant, another possible function is to count the number of ‘set’ bits in the device window. This is the approach we use to interpret system device window values since the algorithm only requires knowledge of whether a device has been active in the recent past and not when in the past it was used. In this case the maximum value of a window is equal to the number of bits associated with it, or its length l .

4.3 Device Windows with Source Annotation

Our EESI process scheduler utilizes device window information to determine scheduling priority. In this section, we investigate using annotations as the update mechanism for process device windows. Annotations convey the following information to the kernel:

- An identifier used to uniquely identify a particular annotation within a process
- An identifier denoting the I/O resource that the annotation pertains to

Annotations are automatically inserted into the source code by an annotator, which can either be integrated into the compiler, or used as a precompilation tool. Annotations are of two types—ones that precede a resource usage and ones that follow it. An annotation describes the resource that will be used next in the former case, and the resource that was just used in the latter.

Our annotator design stems from the observation that applications need to access devices through system calls, which are then managed in kernel space. System calls are typically called through wrapper functions that prepare the arguments to the actual call. For example, the `fread()` function, which is part of the `stdio` library, ultimately calls the `read()` system call. The annotator looks for such library calls, that ultimately result in system calls which may use devices, and inserts annotations both before and after each call occurring in the source code.

We have implemented the annotator as a Perl script that takes source files in C as input. It inserts annotations according to the rules discussed to produce an annotated version of the code as the output. The modified source is then compiled in the usual manner to produce an executable binary. The overall process is illustrated in the Figure 4. The annotator has a list of standard function calls that make system calls and the devices corresponding to the system calls. For instance, the `socket()` function is associated with a network device. The annotator compares the source code with the list of functions, line-by-line, to detect any occurrences. If there is a match, the annotator adds the annotation in the form of a `resourcehint()` call (which is defined as `void resourcehint(int, int);`). This function call takes in a unique annotation identifier as the first argument (implemented as a simple counter in the annotator) and an integer denoting the device corresponding to the system call under analysis as the second.

The decision to keep our annotation scheme simple has been intentional. In addition to minimizing implementation overheads, a simple approach permits one to automatically insert annotations directly even into the executable. Since the unique annotation identifiers merely serve to establish application context in kernel space, the same effect can be achieved by examining the stack during a system call. Our future work aims to achieve this so that limitations of our current approach, namely source code availability and the need for recompilation, can be avoided for legacy applications.

The `resourcehint()` call interfaces with the kernel as a system call and provides the annotation and resource identifiers. A direct-mapped *annotation cache* is also maintained in the kernel and is updated during every `resourcehint()` call. As discussed before, annotations can signal an upcoming device call, or the completion of the call. We term the former an ‘entering’ annotation and the latter a ‘leaving’ annotation. Thus, an entering annotation signals the beginning of potential device usage, and a leaving annotation signals its end. The time elapsed between a leaving annotation and the next entering annotation that uses the same device indicates how frequently a process accesses a device (we do not keep track of the time between an entering annotation and a leaving one as it is not useful for our requirements). This information is collected by storing the current value of `jiffies` during every leaving annotation and comparing it to the current value of `jiffies` during the next entering annotation that uses the same device. Existing estimates of inter-access times for a given device and process are updated using an exponentially weighted moving averaging (with $\alpha = 1/2$) as shown in Equation 4. These estimates are also maintained in the annotation cache.

$$t_{estimate} = \alpha t_{estimate} + (1 - \alpha)t_{observed} \quad (4)$$

Using all of this information, in a static scheme, process device windows can be updated whenever information is obtained from annotations. The resulting update algorithm is shown in Algorithm 2. In the algorithm, p denotes the current process and l denotes the size of the device window in

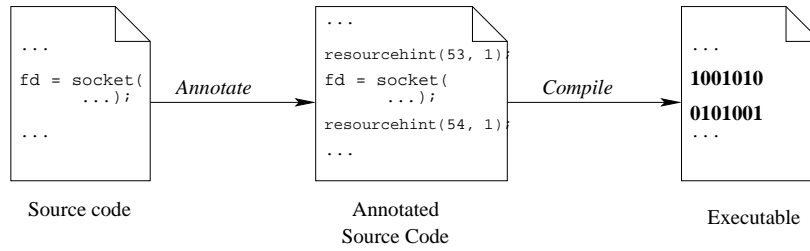


Figure 4: Source Code Annotator

bits. The process device window of a device associated with the calling `resourcehint` is possibly set to zero or maximum based on whether it is a leaving or entering call. An entering call indicates that the device will be used shortly, so we preemptively maximize the process device window value. Similarly, we would need to set the process device window value to zero on a leaving call, since the device is not likely to be used. However, if the device will be used after a short while, zeroing the device window value preemptively will be detrimental. Hence we selectively set it to zero by comparing the expected next use of the device (based on history) against a threshold time value (dependent on the device). For other devices, a similar check is performed to decide if the device window should be set to zero. For the results in this paper, we use a constant threshold value of two clock ticks (`jiffies`) for all devices. Figure 5 summarizes the high level design of the annotation based system.

ALGORITHM 2.

During each `resourcehint(uid, dev)` call,
 \forall devices d s.t. $d \neq dev$,
if $next_use(d) > threshold(d)$ then set $W_{d,p} = 0$
In case of a leaving `resourcehint(uid, dev)` call,
if $next_use(dev) > threshold(dev)$ then
set $W_{dev,p} = 0$
In case of an entering `resourcehint(uid, dev)` call,
set $W_{dev,p} = 2^l - 1$

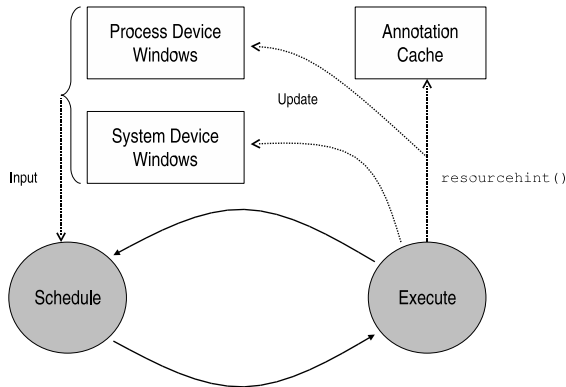


Figure 5: Device Windows with Source Annotations

4.4 A Collaborative Approach

A history-based approach and an annotation-driven scheme both have their own benefits. We attempt to combine both solutions to create a good prediction mechanism for our

EESI scheduling algorithm. The introduction of compile-time hints allows process device windows to more rapidly adapt to changes in device usage patterns. In other words, when a process stops using a device, with the dynamic algorithm, the device window takes a non-negligible time to reduce to zero value since any set bits must be shifted out. On the other hand, annotations can provide enough information to immediately clear a window. Similar behavior is observed when a process starts to use a device. However, as previously mentioned, a static method fails when a system call selectively uses a device. This scenario occurs, for instance, when an application writes to the disk. The actual write doesn't occur until the dirty buffers are flushed by the kernel. A scheme that relies on annotations alone to set device window values will incorrectly predict device usage.

To avoid these ‘false positives’, we allow an entering annotation to assign the maximum value to the device window, but also check if the device is subsequently used. If it turns out that the device is not actually used (this check is done at the corresponding leaving annotation), a flag in the annotation cache is set so that the next time the annotation is encountered, the maximum value is not assigned to the device window. In this case, the device window is set using dynamic device usage information only. A leaving annotation, on the other hand, doesn't present such a problem since it is expected that a device will not be used outside the system call.² In other words, we combine the two approaches by allowing both mechanisms to update process device windows simultaneously. If it is found during execution time that a particular entering annotation is ineffective, it is subsequently ignored and the system relies solely on dynamic updates to predict device usage in that area of the application. Figure 6 presents the design of this collaborative update mechanism used in our EESI system.

5. EXPERIMENTAL EVALUATION

We evaluate our EESI system using a Linux implementation on an XScale-based evaluation platform from Intel. In particular, we modify the 2.4.27 Linux kernel. These modifications include the insertion of device windows state and the annotation cache. The standard `goodness()` function in the Linux scheduler is changed to incorporate the value of $M(p)$ into its calculations in order to prioritize processes accordingly. Other modifications include trivial kernel additions for device usage monitoring.

²Buffering can sometimes lead to such a scenario, in which case the dynamic update can set the appropriate device window value

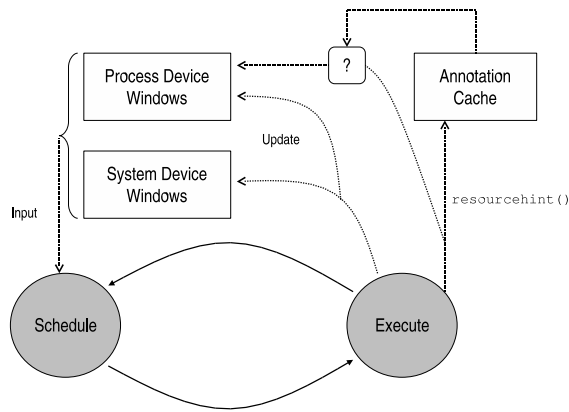


Figure 6: EESI Device Window Update System

5.1 Sitsang Evaluation Platform

The Intel Sitsang evaluation platform is designed around the PXA255 processor. The PXA255 integrates the Intel XScale microarchitecture with various controllers and peripherals including a memory controller, universal serial bus support, and a DMA controller. Additional features include slots for CompactFlash and Secure Digital memory cards. We utilize the CompactFlash slot for our 802.11 network card (Linksys WCF12), and a USB port for a flash storage solution.

We choose to utilize the Sitsang platform in our experiments for various reasons. First, the board is representative of future ‘high end’ embedded platforms such as cellular phones. Such platforms are intended to replace what are now multiple devices carried by end users like PDAs, cell phones, or calculators into single, multi-function devices able to carry out a wide variety of tasks. The XScale processor is also an energy efficient core with dynamic frequency and voltage switching capabilities. The Sitsang combines all of these attributes with a PDA-like form factor.

To test the ability of EESI to provide energy benefits for timeout based systems, we attach a wireless network card and a flash device to the Sitsang. For the wireless card, we enable the normal 802.11 sleep mode. The Sitsang is also designed to provide the ability to toggle power to various subsystems. Since the flash device is powered from the USB bus, we use this feature to implement a software based timeout mechanism. This mechanism powers down the flash device whenever it is idle for 100ms. Future embedded platforms will likely be designed for aggressive power management of subsystems similar to this approach.

5.2 Application Scenarios

Experiments utilize a variety of application mixes to evaluate the ability of EESI to obtain savings. All test scenarios include background computational tasks that do not require any device I/O. For communication-bound processes, we use four different applications. The first is a synthetic application designed to mimic periodic network applications. For many application domains, such as multimedia or robotic software where local sensor data is manipulated and transmitted, there is periodic communication activity where each transmission is preceded by some computation. To generally

analyze this application class, we execute a synthetic application which periodically computes and transmits data. Scenarios 1, 2, and 3 include two instances of this application with increasing device usage rates (relatively 1x, 5x, and 50x).

We also use actual network application scenarios. These applications include a JPEG compression application which processes PPM images and transmits the compressed results. We also develop a scanner application that imitates the transmission of data resulting from label scanning, a scenario common in inventory or processing applications. Finally, we use an audio playback application, Dreamplayer, to stream WAV files over the network. Our 4th experimental scenario includes the JPEG and scanner applications, and scenario 5 utilizes the JPEG application along with streaming audio.

To test EESI on another I/O device, and with the use of multiple devices, we include the use of flash storage. Scenario 6 consists of audio playback from a file stored on the flash device. Scenario 7 extends this mix by including JPEG compression and transmission, where the PPM image files are stored locally on flash and the resulting JPEGs are transmitted over the wireless link. These scenarios are summarized in Table 2.

5.3 Evaluation Results

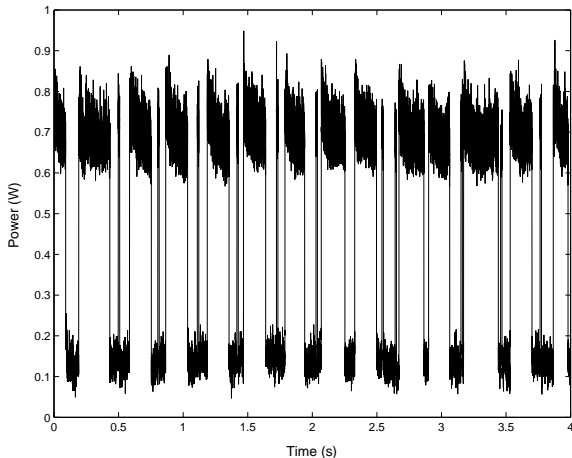
To evaluate the EESI system, we perform power measurements of I/O devices during the various execution scenarios. For scenarios (1)-(5), we measure the power consumed by the wireless card, and for the last two scenarios, we monitor the power usage of the USB flash storage device. To prevent disturbing the supply voltage to these devices during measurements, we implement a measurement circuit based around the SCD10PUR current sensor IC (integrated circuit). The integrated circuit is placed in series with the supply voltage line, and outputs a voltage proportional to the current passing through the inputs. This output is then amplified using op amp circuitry and measured using an oscilloscope. We sample both supply voltage and the output of the measurement circuit at a 4kHz sampling rate. The oscilloscope is capable of buffering 4 seconds of data at this rate. All measurement values are obtained by calculating averages over a total of 60 seconds of data.

To illustrate the ability of EESI to burst accesses and provide extended idle periods, we provide the power signature of the network card during the execution of scenario three in Figure 7. The figure clearly shows that EESI is able to perform access bursting and simultaneously provide longer sleep times. The isolated spikes during sleep periods are due to normal client transmissions to the access point at 100ms intervals.

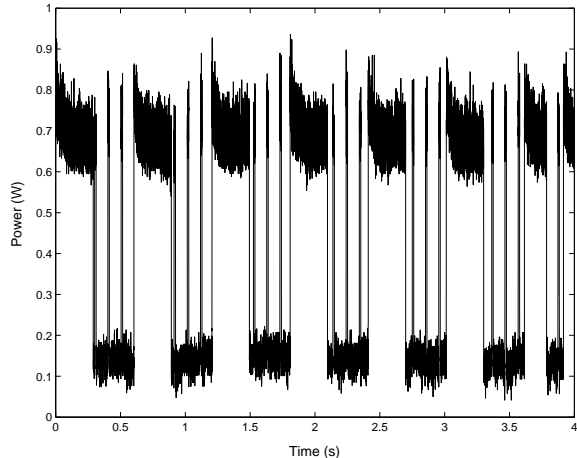
Figure 8 lists the experimental results for various scenarios. Figure 8(a) gives the average energy consumption (power) of the device being measured, and Figure 8(b) illustrates the benefits in time spent in low power mode. We see significant savings, up to 18%, in average power. Since processes are allotted the same amount of service with EESI as they are in the default Linux system, this directly relates to energy savings. Similarly, benefits of up to 17% in time spent in

Table 2: Experimental Scenarios

Scenario	Applications Used
1	Periodic network usage applications (1x)
2	Periodic network usage applications (5x)
3	Periodic network usage applications (50x)
4	JPEG compression and data scanning
5	JPEG compression and audio streaming
6	Audio streaming from USB flash
7	Audio streaming and JPEG compression with data on USB flash



(a) Default Scheduler



(b) EESI

Figure 7: Network (802.11) Power Signatures

low power mode are obtained. We observe that for scenario five the benefits are not as significant as the others. This is due to the fact that the audio streaming application acts as a data sink as opposed to a data source. Since a read access to a network device is an asynchronous operation, local scheduling changes are not as impactful in modifying actual device activity periods. Overall, though, our experimental results illustrate that our EESI system can provide significant I/O energy benefits for an embedded platform.

6. CONCLUSIONS AND FUTURE WORK

Reducing the system-level energy consumption of embedded devices is an important and critical design element of these types of platforms. In this paper we present EESI, a system that combines dynamic operating system state with compiler annotations to provide enhanced power characteristics of I/O devices while providing equivalent service to the default Linux scheduler. We incorporate two data structures into the kernel, namely device windows and an annotation cache. These data structures are utilized by our infrastructure to provide an energy enhanced process scheduler. Our experimental results illustrate that EESI can significantly improve the energy consumption of I/O devices.

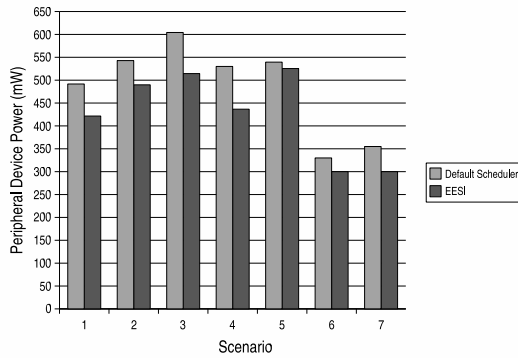
The research presented in this paper will be extended along multiple avenues in our future work. First, although this paper attempts to optimize the power consumption of peripheral devices, it does not address the direct savings that can be obtained from the processor itself. We will address

this by integrating DFS/DVS with our EESI infrastructure, thereby providing a holistic system-level approach to energy management.

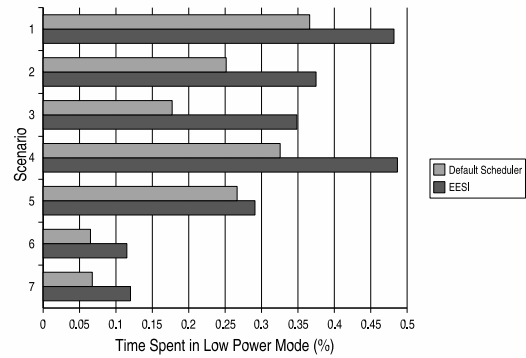
In terms of our use of compiler-driven information, in addition to hints for efficient DFS/DVS, we plan to extend our system to provide improved performance from I/O devices. This includes using our inter-access device access estimates to preemptively wakeup devices to reduce delays experienced by applications. We also plan to utilize compiler hints for wakeup, and study how the aggressiveness of such an approach scales with power/performance savings. Finally, we want to interface our compiler hints with intelligent data readahead to allow for improved I/O power savings by buffering data into memory and shutting down the particular device.

7. REFERENCES

- [1] N. AbouGhazaleh, D. Mosse, B. Childers, R. Melhem, and M. Craven. Collaborator Operating System and Compiler Power Management for Real-Time Applications. *In Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, May 2003.
- [2] F. Bellosa. The Benefits of Event-Driven Energy Accounting in Power-Sensitive Systems. *In Proceedings of the 9th ACM SIGOPS European Workshop*, September 2000.



(a) Average Energy Consumption



(b) Time Spent in Low Power Mode

Figure 8: Experimental Results

- [3] P. Chou, J. Liu, D. Li, and N. Bagherzadeh. IMPACCT: Methodology and Tools for Power-Aware Embedded Systems. *Kluwer Design Automation of Embedded Systems*, October 2002.
- [4] C. Ellis. The Case for Higher-Level Power Management. In *Proceedings of the Seventh IEEE Workshop on Hot Topics in Operating Systems (HotOS-VII)*, March 1999.
- [5] R. Kravets and P. Krishnan. Application-Driven Power Management for Mobile Communication. In *Proceedings of the Fourth ACM International Conference on Mobile Computing and Networking (MOBICOM)*, pages 263–277, October 1998.
- [6] H. Li, C. Cher, T. Vijaykumar, and K. Roy. Vsv:l2-miss-driven variable supply-voltage scaling for low power. In *Proceedings of the IEEE International Symposium on Microarchitecture (MICRO-36)*, December 2003.
- [7] Y. Lu, L. Benini, and G. Micheli. Low-Power Task Scheduling for Multiple Devices. In *8th International Workshop on Hardware/Software Codesign*, pages 39–43, 2000.
- [8] Y. Lu, L. Benini, and G. Micheli. Operating System Directed Power Reduction. In *International Symposium on Low Power Electronics and Design*, pages 37–42, 2000.
- [9] Y. Lu, E. Chung, T. Simunic, L. Benini, and G. Micheli. Quantitative Comparison of Power Management Algorithms. In *Design Automation and Test in Europe*, pages 20–26, 2000.
- [10] R. Nathuji and K. Schwan. Reducing System Level Power Consumption for Mobile and Embedded Platforms. In *Proceedings of the International Conference on Architecture of Computing Systems (ARCS)*, March 2005.
- [11] C. Poellabauer and K. Schwan. Power-Aware Video Decoding using Real-Time Event Handlers. In *Proceedings of the 5th International Workshop on Wireless Mobile Multimedia (WoWMoM)*, September 2002.
- [12] C. Poellabauer and K. Schwan. Energy-Aware Traffic Shaping for Wireless Real-Time Applications. In *Proceedings of the Real-Time and Embedded Technology and Applications Symposium*, May 2004.
- [13] E. Shih, P. Bahl, and M. Sinclair. Wake on Wireless: An Event Driven Energy Saving Strategy for Battery Operated Devices. In *Proceedings of ACM MobiCom*, pages 160–171, September 2002.
- [14] M. Squillante and E. Lazowska. Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling. In *IEEE Transactions on Parallel and Distributed Systems*, 4(2):131–143, February 1993.
- [15] V. Swaminathan, K. Chakrabarty, and S. Iyengar. Dynamic I/O Power Management for Hard Real-time Systems. In *Proceedings of International Symposium on Hardware/Software Codesign*, pages 237–243, 2001.
- [16] V. Swaminathan, C. Schweizer, K. Chakrabarty, and A. Patel. Experiences in Implementing an Energy-Driven Task Scheduler in RT-Linux. In *Proceedings of the Real-time and Embedded Technology and Applications Symposium*, pages 229–239, 2002.
- [17] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for Reduced CPU Energy. In *Proceedings of the First Symposium on Operating Systems Design and Implementation*, pages 13–23, November 1994.
- [18] A. Weissel, B. Beutel, and F. Bellosa. Cooperative I/O-A Novel IO Semantics for Energy-Aware Applications. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI)*, December 2002.
- [19] H. Zeng, C. Ellis, A. Lebeck, and A. Vahdat. Currentcy: A Unifying Abstraction for Expressing Energy Management Policies. In *Proceedings of USENIX*, pages 43–56, June 2003.
- [20] H. Zeng, X. Fan, C. Ellis, A. Lebeck, and A. Vahdat. ECOSystem: Managing Energy as a First Class Operating System Resource. In *Proceedings of the Tenth International Conference on Architectural*

*Support for Programming Languages and Operating
Systems (ASPLOS X), October 2002.*