Energy-Efficient Device Scheduling through Contextual Timeouts *

Balasubramanian Seshasayee and Karsten Schwan
Center for Experimental Research in Computer Systems
Georgia Institute of Technology,
Atlanta, GA 30332, USA
{bala, schwan}@cc.gatech.edu

Abstract

Handheld and embedded hardware platforms are operating with an increasing number of internal and external devices, potentially increasing energy consumption and more importantly, motivating the need for energy management techniques for peripheral devices. This paper presents a platform-wide, system-level approach to dynamic energy management, termed *contextual timeouts*. The approach exploits the fact that most current peripheral devices support the ability to switch to a low power mode when not in use and automatically resuming operation upon use. The approach utilizes the energy savings derived from such device suspensions, considering that the device suspend/resume actions themselves consume power and have associated latencies. Contextual timeouts do not require programmer involvement. Instead, dynamic instrumentation is used to automatically capture and monitor the *contexts* (i.e., the execution points) at which programs make the service requests that cause device usage. From such dynamic monitoring data, system-level algorithms predict future request times and manage devices to best meet program needs under predicted behaviors. Adaptive methods for dynamic workload characterization coupled with runtime techniques for request prediction result in experimentally obtained energy savings of up to 50% *over* an aggressive timeout-based regime on a Linux-based iPAQ PDA.

1 Power Management for Peripheral Devices

Peripheral devices account for a sizable and increasing portion of the overall power consumption of embedded platforms. For example, as discussed in [24], in a Toshiba 410 CDT, roughly 72% of the power budget is accounted for by the peripherals (display, hard disk and wireless card). An inherent property of embedded systems, therefore, is the ability to perform dynamic power management. Management techniques include timeout-based methods for peripherals operating at the driver level [2] or higher level methods that assume program-specified knowledge, obtained either through profiling [14] or explicitly disclosed by the programmer [11, 1]. Other techniques have relied on modifying scheduling policies [15] to aggregate tasks so as to increase the idle periods of devices. Yet others compute the I/O schedule from a predetermined task schedule in hard real-time systems[23]

This paper presents a platform-wide, system-level approach to dynamic energy management, termed *Contextual Timeouts* (CT). CT uses program instrumentation to automatically capture and monitor the dynamic *contexts* (i.e., the execution points) at which programs make the service requests that cause device usage. Such runtime information is used to predict future request times, and those predictions are then used to better manage peripheral devices. Energy management decisions in CT are based on the break-even time of a device [8]. That is, CT decisions determine whether energy-wise, it is more beneficial to leave the device in the current mode or to suspend it. Note that break-even time is device-dependent, since it depends on factors like the energy needed to suspend/resume the device and the power savings attained in its suspend mode. For example, an iPAQ with the microdrive used in our experiments takes roughly $2.1J$ for a suspend/resume cycle, and the difference in power between idle ($1.66W$) and suspend modes ($1.61W$)

*This work was funded in part by NSF-ITR award and Intel corporation

is about $0.05W$. The total time duration for a suspend/resume cycle is about $1s$. This translates to a break-even time of $\frac{2.1J-1.66W\cdot 1s}{0.05W}$, or roughly 9 seconds.

The CT approach to reducing the power consumption of peripheral devices (1) utilizes the energy savings derived from driver-level device suspensions, assuming that devices expose their different power modes to higher system levels, it (2) takes into account that device suspend/resume actions themselves consume power and have associated latencies, and it (3) operates without program involvement or higher level specifications about anticipated program behavior. In this paper, we demonstrate the efficacy of CT on a Linux-based iPAQ PDA, showing energy savings of up to 50% over an aggressive timeout-based regime. In future work, CT will be used in conjunction with power management methods for the processor cores of embedded platforms [12, 26], but we have not yet investigated the joint effects of processor and CT-based peripheral management. CT can also be used to improve the efficacy of purely system-level techniques for reducing the power consumption of peripheral devices, like the *device windows* developed in our own earlier research [15].

The remainder of this paper first describes the notion of contextual timeouts, followed by an explanation of their implementation for a Linux-based iPAQ handheld platform. A comparison to related work (Sec. 6) demonstrates the novel nature of the approach. Representative applications and an adaptive method for classifying their workload behavior are discussed in Sec. 4. Microbenchmarks and evaluations with these workloads are described in Sec. 5. Conclusions and a description of future work appear in Sec. 7.

2 Contextual Timeouts

2.1 Basic Concept

Timeouts are the common way of managing the power consumption of peripherals. Commonly realized in the device driver, this technique operates under the assumption that when none of the processes require the use of a device for some period of time, it is likely that none of them will need to use it in the near future (i.e., temporal persistence). Unfortunately, timeouts operate without contextual information about current process behaviors. This causes violations of the temporal persistence assumption, with the result that timeouts may be ineffective, and in some cases, even deletirious to platform energy consumption (particularly when using aggressive management techniques).

The core idea behind *contextual timeouts* (CT) is to timeout and suspend devices based on the current *contexts* of the executing user-level processes. For the purposes of this paper, a *context* is simply defined as a specific point in a program that may generate a system call. The following code snippet illustrates the idea (the contexts are marked with comments) for the `read` and `fread` calls in an application program. The `read` system call clearly constitutes a program context. `fread` is labelled as a second context since it 'may' translate to a `read` call.

```
...
do {
  i = read(fd, buf, count);  /* Context 1 */
  count += 10;
  fread(buf, count, 1, fp);  /* Context 2 */
} while(...);
...
```

This simple example establishes two important properties of the contexts used in CT: (1) a program context simply establishes that a system call is possible at some execution point in the program, and (2) except for synchronous system calls and device accesses, context execution does not deterministically imply device access. Nonetheless, as shown later, even for (2), it is possible to invoke device power mode management with contexts, involving a timeout decision at each context after computing the expected next use of the device.

Contextual timeouts have an inherent advantage over conventional timeouts in that, if device idle periods are known to be very long, the device will be suspended immediately rather than having to wait for its assigned timeout duration. Moreover, when the device's idle period is only slightly larger than its timeout value, the use of conventional timeouts could cause negative energy savings. This happens when the time spent in low power mode is less than the device's break-even time. This is explained by the following analysis.

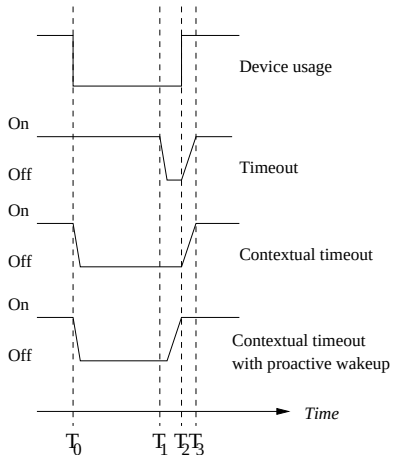


Figure 1: A Comparison of Various Approaches

Figure 1 shows an example of a process that ceases to use a device at time T_0, and resumes its use at time T_2. With a conventional timeout policy, the device will stay on till the timeout expires (say at T_1), at which point it will be put in the low power mode. Once the process issues a service request to the device, though, it takes some time for the device to be "woken" up, and this introduces a latency $(T_3 - T_2)$. This latency also exists with CTs, but the idle time is much larger since CT doesn't wait before switching the device to a low power mode (if the estimated idle time is expected to exceed the break-even time, the device is suspended immediately). The device energy savings from CTs is thus more than that of timeouts by $P \cdot (T_1 - T_0)$, where P is the device power drawn under normal mode in excess of that under the low power mode. Additionally, when the break-even time is larger than $T_3 - T_1$, negative energy savings will be observed from timeouts. If the break-even time happens to be even more than $T_2 - T_0$, CTs will not even suspend the device with a correct prediction. The extra latency incurred with CTs can be avoided by resuming the device to normal mode ahead of its use, taking the resume latency into account. This scenario is illustrated last in the figure, where the device idle time is reduced, but the latency is avoided.

The remainder of this section explains the three basic elements of the CT implementation, which are (1) obtaining and representing the contexts of the executing processes, (2) monitoring device usage under the obtained execution contexts, to enable predictions of future usage, and (3) using the predictions to decide whether devices need to be suspended. Explanations initially assume synchronous (blocking) system calls and device accesses and are then refined to take into account asynchronicities. These elements are depicted in Fig. 2, where contexts are shown as small shaded boxes in executing processes, context representations and monitoring information are stored in a system-level context cache, and decision methods are realized for each device for which context information is maintained.

2.2 Obtaining the Context

The current context of a process is determined by the current point in the program being executed. Since we are interested only in the devices used by a process and since device accesses are mediated by the operating system through system calls, we determine processes' contexts by monitoring their system calls. We have implemented two different approaches, both of which are described below.

In the first approach, annotations inserted into programs are used to identify contexts, and the source code of the program is run through an annotator before it is compiled. Annotations surround both system calls and standard library functions that invoke system calls like `fread()` [1]. Each annotation carries two identifiers - a unique integer that identifies the annotation and one that identifies the device associated with its target system call. For example, as shown in the code snippet below, a `send()` call is associated with the network device and the annotations are inserted surrounding it.

[1] In the remainder of the paper, we use the term system call to refer to all such cases

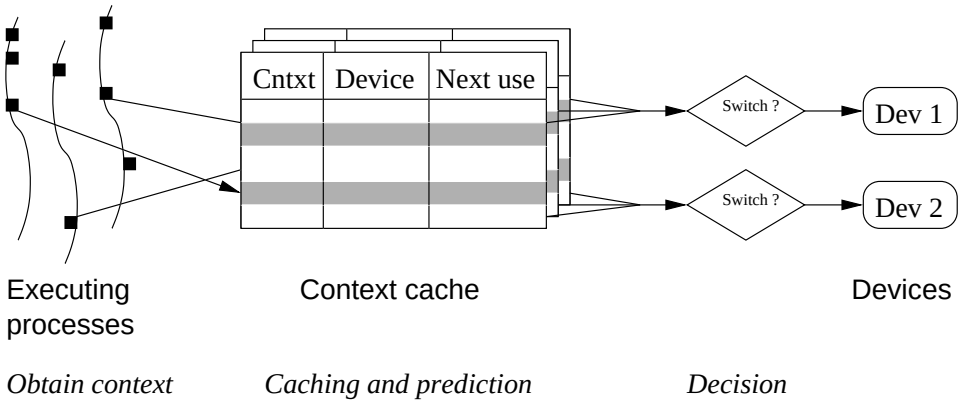


Figure 2: Contextual Timeouts - Basic Architecture

```
...
annotation(23, NET_DEVICE);
send(sockfd, buf, buflen, 0);
annotation(24, NET_DEVICE);
...
```

This implementation realizes annotations via additional 'annotation' system calls, used to convey the unique context identifier and the associated device, to the operating system kernel. Since the annotations are inserted automatically at compile time, programmer intervention is unnecessary, but the approach requires both source code availability and the recompilation of application programs.

The second approach eliminates the need for program sources, program recompilation, and additional system calls, by identifying contexts from 'within' system calls. This involves 'wrapping' system calls in the kernel with annotation functionality that performs the following actions: (1) it finds the calling process' call stack from the current stack pointer, (2) it computes the unique context identifier for each context by summing all current call stack return addresses. As with the first implementation, this approach does not involve programmer or user intervention. In addition, the context obtained is more accurate. Specifically, with annotation-based approach, two different points in the program may be identified as the same context depending on how the annotated function is called. For example, consider the case when all `read()` calls in a program are made through a wrapper function, as shown below:

```
int myread(int fd, char *buf, int size)
{
    ...
    annotation(15, DISK_DEVICE);
    retval = read(fd, buf, count);
    annotation(16, DISK_DEVICE);
    ...
}
```

```
void func1 (void)
{
    ...
    x = myread(fd, buf, cnt);
    ...
}
```

```
void func2 (void)
{
    ...
    y = myread(fd, buffer, count);
    ...
}
```

The annotations are now inserted inside the wrapper function. The actual context of the call is now lost, since two distinct locations that call this wrapper function (at `func1` and `func2`) appear as the same context in the kernel, with the annotation-based approach. With stack-based approach, the two contexts

get distinct context identifiers due to differences in the call stack. But the call stack approach requires that all device-related system calls be wrapped, resulting in additional changes to the kernel.

This paper uses the annotation-based approach for most of the experimental results shown in Sec. 5. Feasibility of the call stack-based approach and the performance differences inherent in the two approaches are evaluated with a realization of the call stack-based approach for the `connect` system call.

2.3 Prediction

Prediction is based on dividing each process context into an 'entering' and a 'leaving' context, corresponding to the beginning and the end of the associated system call. With the annotation-based approach, this simply implies inserting annotations before and after the system call. With the stack-based approach, the same effect is achieved by setting an extra flag when the system call is entered and resetting it just before the system call returns. The entering and leaving contexts demarcate the period during which the device would be needed. Note that this is violated by non-blocking calls and when system calls are buffered, as discussed in further detail in Sec. 2.5.

The goal of the prediction module is to determine, at each context, when the associated device will be used next, based on the information maintained for that context. Given some leaving context, the time duration for the next entering context for the system call associated with the same device is the device's expected time of next use. For simplicity, entering and leaving contexts are treated in the same fashion, where for an entering context, the time to next device usage degenerates to zero (i.e, the entering context itself signifies device use).

The core data structure used by the prediction module is the context cache. The context cache is a per-process, per-device cache used to cache the contexts encountered in the executing process. It is implemented as a direct-mapped cache, indexed by a hash-value of the context identifier. Each cache entry contains, apart from the context identifier, the running estimate of the expected duration till the next access of the device ($T_{next\ use}$), and the recent measurement of this quantity ($T_{measured}$).

The algorithm used for prediction is illustrated by the flowchart in Fig. 3. It proceeds in three steps. First, the cache entry corresponding to the current context identifier is created, if it doesn't already exist. Next, the value of $T_{measured}$ is calculated as follows:

A leaving context sets $T_{measured} =$ `jiffies`

An entering context sets it to $T_{measured} =$ `jiffies`$-T_{measured}$

thus storing the time elapsed between a leaving context and the next entering context. Finally, during a leaving context, $T_{next\ use}$ is updated based on the recent estimate ($T_{measured}$), using exponentially weighted moving average (EWMA)–i.e., $T_{next\ use} = \alpha \cdot T_{next\ use} + (1 - \alpha) \cdot T_{measured}$. Of course, for an entering context, $T_{next\ use} = 0$. EWMA is chosen over other time series estimators like ARIMA for the flexibility and the low overhead it provides. Note that the same approach is used in [8] to estimate idle periods. In our implementation, we use an α value of 0.5, but this can be changed to lower or higher values for faster or slower responses to changes. The value of $T_{next\ use}$ is now used as the idle period estimate in the decision module.

2.4 Decision

The decision module runs when the execution of an entering/leaving context has updated the context cache. The decision module determines whether any of the devices should be suspended. The algorithm used is straightforward. In each leaving context, for each device not already in low power mode, if $T'_{next\ use} > T_{break-even}$, the device is immediately suspended, otherwise no action is taken. The only quantity needed for the decision, therefore, is $T'_{next\ use}$. This is computed from the $T_{next\ use}$ values of all the processes, adjusted for the time elapsed from their respective contexts, to find the time period left for the earliest access of the device by any process. Also, in each entering context, if the device is in low power mode, it is switched back to normal mode. The effects of these actions are simply that (1) in leaving contexts, a device is turned off when such a decision is deemed beneficial, and (2) in entering contexts, a device is turned on as soon as possible, even before the system actually accesses the device. A predictive method[8] can be used to

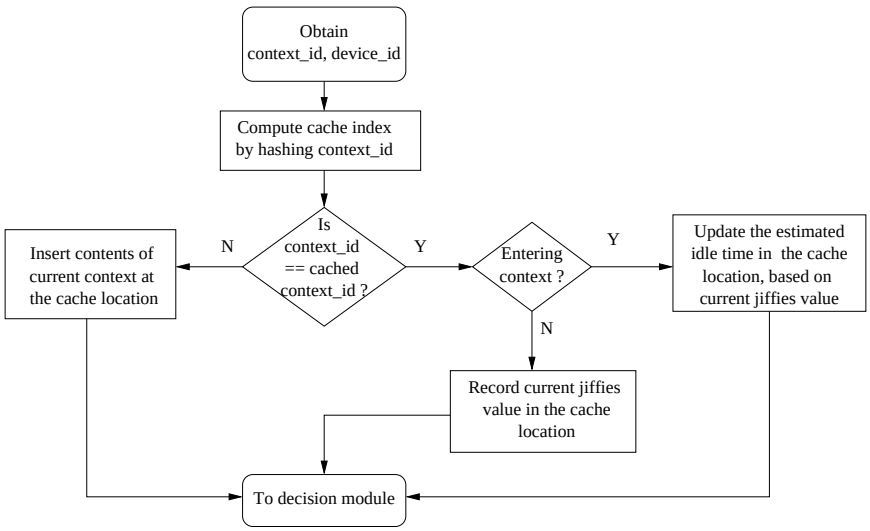


Figure 3: Flowchart Describing the Prediction Procedure

resume a device from suspended state after a time period $T_{next\ use}$, making the "wakeup" decision similar to the timeout one, but since this value is largely dependent on the accuracy of the EWMA model, this was found to be effective only for periodic workloads (discussed in Sec. 4).

2.5 Asynchronous and Buffered Device Accesses

So far we have discussed our technique assuming the ideal case where each device-related system call results in an access to the device and the period of access is limited to within the period of the call. This holds for calls like blocking `recv`, for example. However, it is not true in many other cases, including for non-blocking calls like `send` where the call may immediately return before the device finishes servicing the request, or due to buffering, commonly encountered in file system calls.

Non-blocking calls are handled using timers. For each non-blocking call, a timer is created as the system call returns. The handler for this timer checks to see if the device is done servicing the request, through polling. If the next system call that makes use of the same device is called by the process, the timer is reset and the estimated $T_{next\ use}$ value is set to zero (since the device doesn't get a chance to become idle). Otherwise, we mark an extra field in the cache entry to indicate that the start time is the moment when the device completes the request rather than when the system call returns. Similarly, the decision module is invoked not in the leaving context when the system call returns, but in the timer handler, when the device is known to be done servicing the request. The polling interval is currently chosen to be one second, which is sufficiently large to allow non-blocking operations to finish up their requests in most cases, and cause low performance overheads, yet small enough to not ignore any device idle times. This has been implemented in Linux using the kernel timers. Non-blocking calls are marked so by the annotator by setting an extra field (not shown in previous annotation examples).

Buffering is common in file system calls like `read`, where each system call need not cause a device access. We account for these cases by maintaining additional fields in the cached entities, termed `current_misses` and `expected_misses`. The first quantity tracks the number of misses since the last hit (by hit, we mean a system call that leads to a device access), and is used as a measure of the frequency of the device accesses. The second quantity, `expected_misses`, is used as the running estimate for this quantity. It is replaced by the previous quantity during every device access. Long-running processes that make repeated accesses with some regularity have a constant `expected_misses` value. For instance, a sequential read of a file will cause the buffer cache to fetch the file contents in chunks, and if a constant amount of data is read in

during every `read()` call (which is typically the case), it leads to a constant `current_misses`. Based on this quantity, a fairly reliable value of $T_{next\ use}$ can be obtained using the equation $T_{next\ use}$ = (`expected_misses - current_misses`) · $T_{next\ use}$/`expected_misses` (i.e., the fraction of $T_{next\ use}$ that is remaining, before encountering a hit). Note that this mechanism cannot be used for `write` related calls, since write-related device accesses occur outside such system calls, due to *bdflush* and *kupdated* daemons, depending on the number of dirty buffers in the buffer cache. Device writes are not synchronized with system calls and hence, are treated as events "outside" our control. We minimize the impact of such events by adjusting the *bdflush* parameters and the *kupdated* interval to minimize write frequency. These parameters are modified to reflect the aggressiveness of the power management scheme (which in turn is decided by the time-out values). Alternatively, all write calls to a device can be batched together, as proposed in [19].

3 Implementation

Contextual timeouts have been implemented in the Linux kernel 2.4.19, patched with rmk6 and pxa13. The changes to the kernel involve addition of the `annotation()` system call, which implements the context cache, prediction and decision modules, and support for non-blocking and buffered calls. In addition, the implementation of the buffer cache and the network stack are modified to update the context cache when device accesses complete. Currently, only PCMCIA devices are supported, and `pcmcia_suspend()` and `pcmcia_resume()` are used to switch between the device power modes.

The annotator is a Perl script that operates on C source code to detect the device-related system calls and add the annotations. It also tracks file descriptors to identify the device used. It uses a database (currently 23 entries) containing system call–device relationships, and the attributes of the system call (blocking/non-blocking), to add the appropriate annotations. The call stack approach is implemented on a single system call (`connect`), determining the context before and after the system call, to demonstrate its feasibility. A full blown stack based implementation will have all device-related system calls modified this way.

4 Workloads and Applications

The CT approach targets applications running on battery-powered devices like handheld PDAs or cellphones. Consequently, the application benchmarks chosen to evaluate the approach emulate the typical usage of such devices:

- Web browser - we use a simple *wget*-based WWW browser. The browser makes HTTP requests for HTML and GIF files (less than 20KB), and caches the response pages on the device's hard disk. User traces from [6] are used to drive the requests. As one would expect, the behavior of this application is bursty in nature. Inter-access times vary between 1 second to over two minutes.

- MP3 player - a streaming MP3 player using the *mpglib* library is used to play an MP3 file (128kbps, 44kHz stereo) over HTTP. This application uses only the network device, and it exhibits strongly periodic behavior.

- JPEG viewer - an application making use of *libjpeg*, which reads a few JPEG files (640x480, about 350KB each) from the disk and decodes them. This has the same behavior as the slideshow program commonly found in cellphones and PDAs. The application spends most of its computing time on the decoding portion of the program. We assume the user input to be Poisson-distributed [18].

- Scanner - this application emulates the behavior of scanning devices (e.g., bar code readers) relevant in the handheld application domain. In our case, the data read is transferred to a remote server over the network. Again, user input is Poisson-distributed (λ = 10, each data is 100 chars long, 128KB buffer). However, since buffering is commonly used in such applications, this leads to a smoother, less bursty resource usage pattern.

The applications' resource usages are as outlined in Table 4.

Application	Disk device		Network device	
	Read	Write	Send	Recv
WWW Browser	Minimal	Yes	Minimal	Yes
MP3 Player	No	No	Minimal	Yes
JPEG viewer	Yes	No	No	No
Scanner	No	No	Yes	No

Table 1: Applications - Resource Requirements

Workload	Description	Example
Periodic	periodic use of a resource (ideal case)	streaming audio
Bursty	resource used extensively for a brief period, followed by long periods of inactivity	web browsing
Periodic with jitter	more practical scenario of periodic workload	streaming audio under network load
Random	usage pattern is random	
Random with large idle periods	same as above but idle periods are large	

Table 2: Description of Workloads

To understand the effectiveness of contextual timeouts and compare it with other techniques, it is important to determine the kinds of workloads that are suitable/unsuitable for the different techniques, based on their usage patterns. Consider the simplest case of a single process workload and a single device. Ignoring the cases when the device is always/never used, workloads can be classified into periodic and aperiodic ones, based on their usage patterns. Periodic workloads issue device requests at fixed time periods. Soft real-time applications fall under this category. So do multimedia applications since they typically operate on large data in fixed chunks. Aperiodic workloads can be bursty or random. The former typically occurs when there is user input involved – for example, a mail client is used to send/check mails with periods of inactivity inbetween when the mails are read/organized. The latter is used to model the worst case scenario when idle times cannot be predicted.

Table 2 summarizes the different workloads described above and their characteristics.

Based on this, we classify the above workloads into regular and irregular ones, with the expectation that regular workloads are largely more predictable in terms of their device usage behavior. Periodic workloads naturally fall into the regular workloads category, and so do periodic workloads under jitter. Even among aperiodic workloads, some workloads can fall into the regular category. For instance, if the device access is random but the time intervals between the device accesses are large – more specifically, greater than the break-even time, again, our prediction will determine that the expected idle period is large enough to warrant a power mode switch, however inaccurate the prediction value itself might be. Similarly random accesses whose intervals are consistently below the break-even time can also be classified with regular workloads, since the prediction determines that no power mode switches are possible. Irregular workloads are those whose idle periods vary between values less than and more than the break-even time period.

Adapting to irregular workloads While significant energy savings can be attained from our technique on regular workloads (due to better prediction), the same cannot be said about irregular ones. This is typified by a workload with bursty accesses between long idle periods. In this case, the EWMA based prediction algorithm fails to track the idle periods accurately [29]. Conventional timeouts, however, prove to be very effective in this case, since they can selectively timeout during idle periods and 'stay on' during bursty periods. For this purpose, we adopt an adaptive scheme. The scheme optimistically assumes workloads to be regular and therefore, uses the contextual timeouts. At the same time, a low overhead mechanism is used to track the regularity of the current workload. If found to be irregular, we switch to conventional timeouts. We switch back to contextual timeouts whenever the workload again becomes regular. Regularity is tracked using a 32-bit field `irregular_bitvector` in the cache entry. This field is initially set to zero,

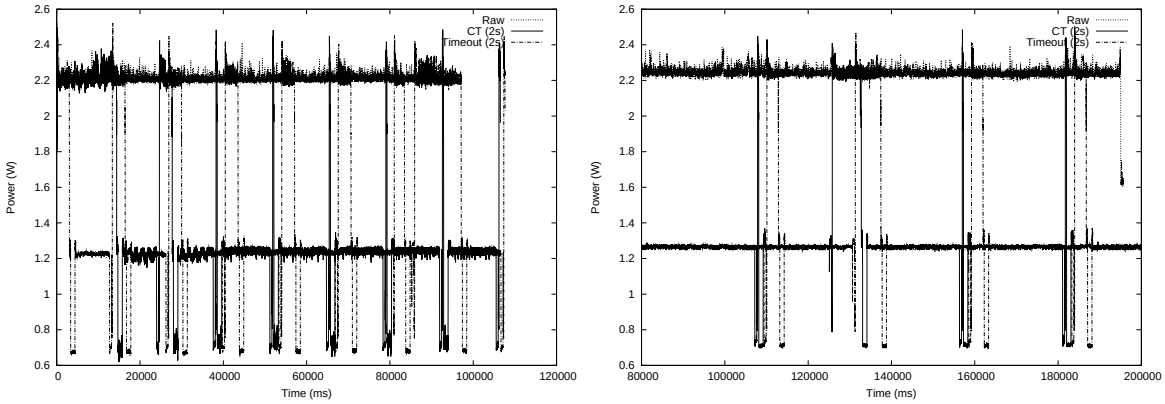
and its most significant bit is set to 1 whenever the $T_{next\ use}$ field is found to switch from a value greater than the break-even time to one less than that or vice versa. Also, this field is shifted right by one bit during every encountered execution context, thus exponentially decaying over time. Therefore, the value of `irregular_bitvector` serves as an estimate of the current workload's degree of irregularity (a higher numerical value indicates greater recent irregularity). This value is compared to a threshold to determine if the workload is sufficiently regular, and consequently, whether contextual or conventional timeouts should be used. The threshold value, again, could be chosen to reflect the desired level of aggressiveness.

5 Experimental Evaluation

Experiments are run on the modified Linux 2.4.19 kernel, on an iPAQ h3800 with a sleeve. Energy measurements are carried out by measuring the voltage and the current drawn by the iPAQ, using an ADC-212 *Picoscope* oscilloscope, sampling at 1kHz frequency.

5.1 Workloads

The first set of experiments are carried out on the set of workloads discussed in Table 2. Each workload is run (1) without power management, (2) with a timeout based and (3) a contextual timeout based scheme. For comparison, we also compute the estimated energy consumption if (4) an oracular power management technique were used. This is obtained analytically by assuming that a device that is in a low power mode is switched to the normal mode in advance by an oracle so that no time is lost in the switching processes, with the end result that the total time consumed in the oracular model is the same as that without any explicit power management. That is, an end user will not be able to perceive any difference between oracular vs. no power management.



(a) Periodic workload

(b) Workload with large random idle times

Figure 4: Regular Workloads

Experiments use an 802.11b wireless PCMCIA card, which has a break-even time of about 1 second. The timeout period for both the conventional as well as contextual timeouts is set to 2 seconds. Experiments were conducted multiple times for each scenario, using the same seed values for random number generators (`rand()`), and variations were found to be negligible. Each run was performed over a period of five minutes. The synthetic workloads were constructed as follows. The workload opens a TCP/IP socket and starts a loop where it receives 100 bytes of data from a server (running in a remote machine), and performs computations on the data, and so on. The amount of computations is varied depending on the type of workload, thus varying the idle times between device accesses. The periodic workload has a period of about 13 seconds. Jitter has this value varied randomly by ±10%. Random workload's idle times are uniformly randomly

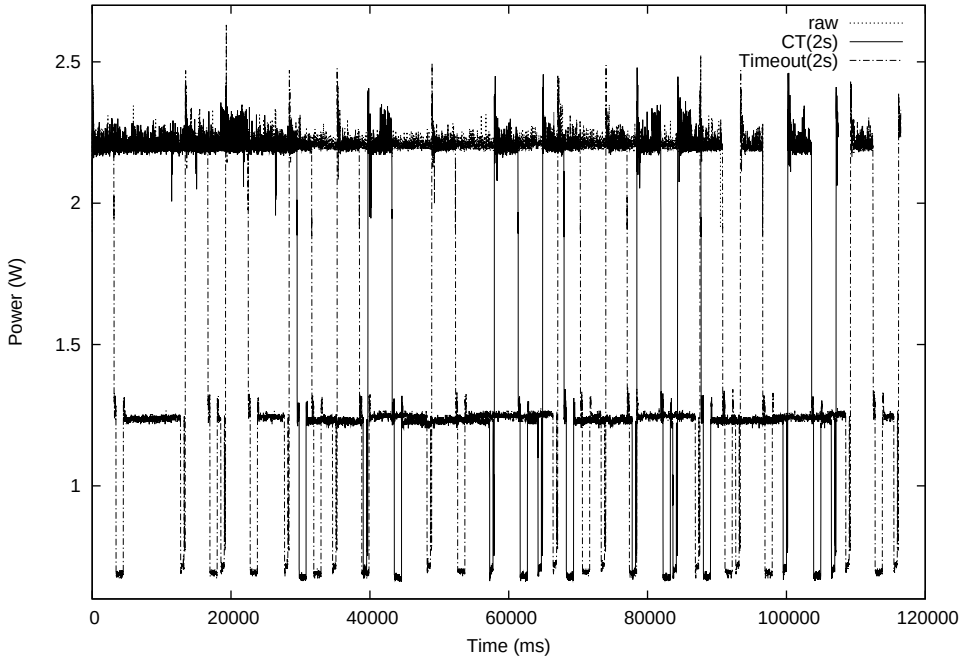


Figure 5: Irregular Workload - Random Idle Times

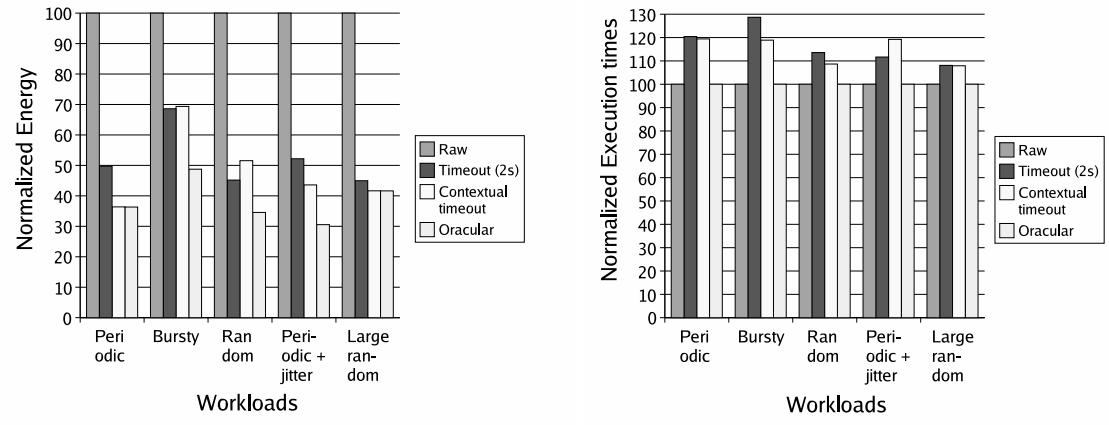
generated between 0-20 seconds. For bursty workload, the idle times were picked from a HTTP trace [6]. Large random idle times are used for the last scenario. The display of the iPAQ is turned off for these measurements.

Fig. 4 shows the power characteristics for a periodic workload and one with large random idle times. In each graph, the power consumption over time with various schemes (no power management, convextual and conventional timeouts) are shown. In the periodic case, the period between device accesses is about 12 seconds. It can be observed that CT doesn't switch the power mode during the first opportunity – this is because the $T_{next\ use}$ value is zero during the initial context. Once this value is estimated, we can observe that in subsequent opportunities it is able to switch power modes immediately. The delay associated with the timeout scheme during each context is also apparent. Similarly, with the next scenario shown, the idle times vary widely, but are always well above the timeout value. In both the cases, we find the timeout scheme to trail the CT scheme in device switches. This is due to the initial power mode switches avoided by CT when it encounters the contexts for the first time, that gives it a head start. However, this difference is not considered in the study for regular workloads, when the execution times for the schemes are compared.

Next, the behavior of the irregular workload (with random idle times) is shown in Fig. 5. CT is shown with a solid line while conventional timeout is shown in dashed line. In this case, CT observes irregularity in the idle times right from the beginning, and hence switches to conventional timeouts immediately. After about 30 seconds, we see CT performing this switch and staying with conventional timeouts (due to high variations in the estimated idle times) throughout. Starting from the 30 second mark, the timeout scheme "lags" behind CT, but except for this lag, the behavior is the same.

The energy consumed and the execution times for the workloads with each of the techniques are summarized in Fig. 6. The normalized energy values shown are the sum of the dynamic energy of the CPU and the energy consumed by the wireless card, scaled such that the said energy consumed is 100 units when no power management is in place. Hence, all values reported are percentage energy values normalized with respect to no power management. Similarly for the execution times. As mentioned before, for the regular cases, the energy and execution times are discounted for the initial "headstart" achieved by CT when it starts to predict the idle times (the numbers for the no power management case were unchanged).

The results indicate that CT matches oracular energy management when workloads are regular. As discussed in Sec. 4, this is the case with the periodic workload and the random workload with large inter-



(a) Normalized energy

(b) Normalized execution times

Figure 6: Results on Workloads

request intervals. On a periodic workload with jitter, CTs do not match the oracular technique, but still perform better than conventional timeouts (by 13%). In this particular case, the presence of jitter causes the inter-request interval to fluctuate between values that are less than and greater than the break-even time. Note that if the jitter caused the intervals to be consistently larger than or smaller than the break-even time, CTs would be able to match the oracle. On the other hand, with bursty and random workloads, the CT approach fails to find any pattern and so reverts to a conventional timeout based approach (causing a 6% negative difference in energy savings). For the bursty and random workloads, the measurements shown were taken over the first few iterations (i.e., around five minutes of execution time), hence we see a difference in energy consumption between the CT and conventional timeouts. Over time, as CT switches to conventional timeout, this difference will disappear. In these experiments, we are able to match the oracle only because no power is consumed by the devices when they are suspended. If some non-zero power is still consumed in the low-power mode, CTs cannot match the oracle, as the extra execution time will account for a corresponding amount of energy due to this residual power.

The execution times for both of the timeout-based techniques are larger than the default case (worst case 20% with CTs and 30% with timeouts), due to the delays experienced in waking up the device. They are almost the same for both on average, but with the irregular workloads, conventional timeout fares better because of fewer "wrong" switches – i.e., switches made when the idle period is less than the break-even time. As mentioned before, over the long run, this difference too will vanish once CT adapts to conventional timeouts after determining that its predictions turn out wrong.

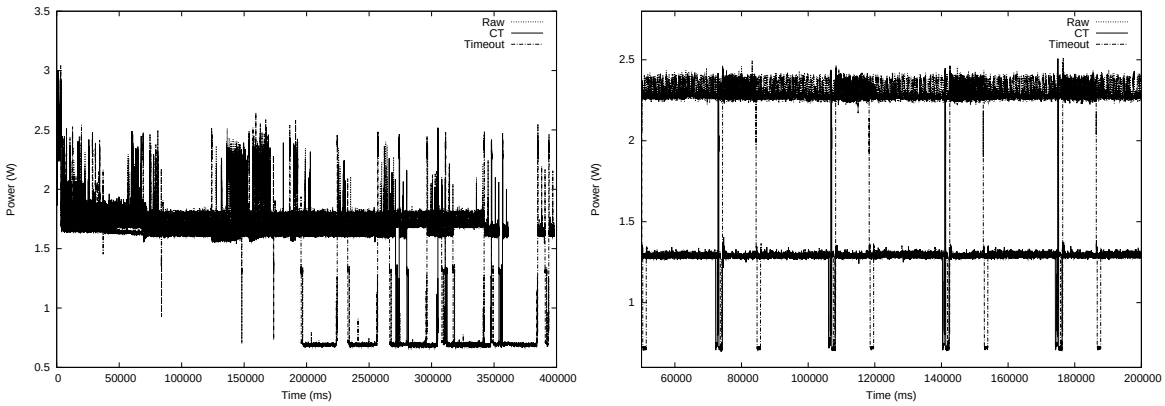
5.2 Applications

The next set of experiments compares contextual timeouts against aggressive or less aggressive timeout mechanisms (realized through small or large timeout values, respectively). Instead of the workloads, we use the actual applications discussed previously in Sec. 4. For this set of experiments, we make use of two devices - a CF-based IDE hard disk [13], connected to the iPAQ sleeve via a CF-to-PCMCIA adaptor, and an 802.11b PCMCIA card [28]. Only the energy consumed by these devices (with the dynamic energy of the CPU) are reported in the results. In addition to the individual applications mentioned in the list, the following combinations are also run:

- MP3 player and WWW browser - a regular and an irregular application,
- MP3 player and Scanner - two regular applications using the same resource, and

- Scanner and JPEG viewer - two applications using different resources.

The contextual timeout and the aggressive timeout each have timeout values set to 2.7 seconds and 11 seconds for the network and disk devices, respectively. In the conservative timeout case, the timeout values are 10 seconds and 15 seconds, respectively. The disk device, an IBM microdrive is found to have a break-even time of about 10 seconds. The high break-even time is due to effective inbuilt power management in the device. For instance, the drive has a hardware-based APM wherein the disk can be spun down immediately after it completes servicing a request. Oracular power management is not included in these results, since unlike with workloads, the device accesses with applications are complex, and computing the oracular behavior is not as straightforward.



(a) WWW browser

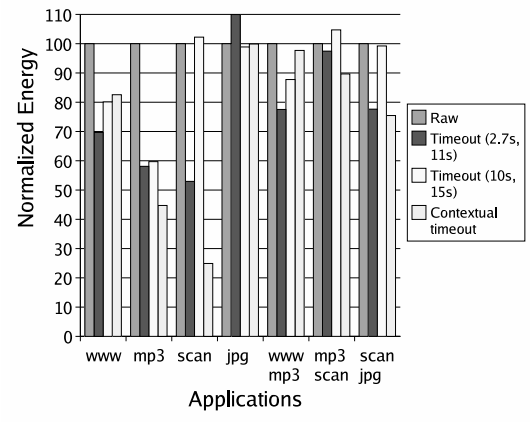
(b) Streaming player

Figure 7: Applications

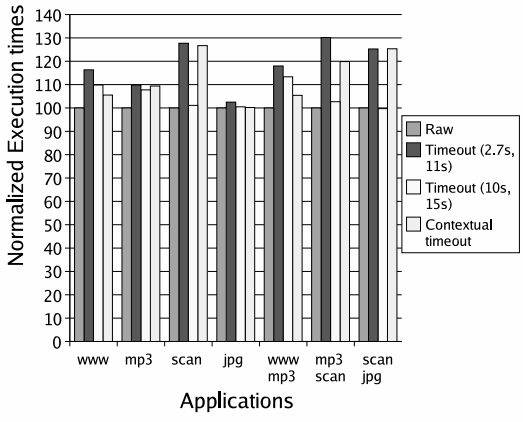
The behavior of the browser and the MP3 player are shown in Fig. 7. The browser is clearly bursty. As with the random workload, CT switches to conventional timeouts after around 200 seconds, after which the behavior of both are identical. The lost opportunities to timeout thus far, causes a difference in overall energy consumption (Fig.8). One can also observe that the disk is being switched to its low power mode in both the cases fairly early, but remains on throughout the experiment with the default case. The MP3 player follows the periodic workload perfectly, as fixed chunks of the file are read and decoded.

Fig. 8 shows the normalized energy and execution times of these experiments. In all cases except those involving the browser, contextual timeouts perform as good as, or better than, the best of the conventional timeouts. The best case scenario occurs with the scanner application, when the contextual timeout approach saves 50% energy over a conventional timeout scheme, which by itself, accounts for a 50% savings over the base case. Due to the buffering, the scanner application is almost periodic, despite the input following Poisson-distribution. The energy savings in this case are even better than the more periodic mp3 player (74% vs 55%) due to the more frequent accesses to the device with the scanner (the average idle time is 35 sec. with the mp3 player and roughly 10 sec. in the scanner). This translates to more opportunities for the CTs, resulting in better energy savings. Note that in absolute terms energy savings are not "better" due to frequent power mode switches. But since CT saves more energy than timeouts for each switch, the relative energy savings increases. Another effect is the negative energy savings observed with a large timeout value, on the scanner application. Since this value is very close to the observed average idle time for the application, the unnecessary power mode switches cause this phenomenon. The same effect is also observed with the timeout scheme in the jpeg application. In all other cases, the savings are not as significant. In fact, we also observe negative energy savings with CTs in the browser scenario as noted before.

With mixed applications, the combined effects of the different accesses can be observed. In the first two cases of mixed applications, the applications share the same device, whereas with the third case they use different devices. Not surprisingly, the energy savings obtained in the first two cases are worse than that



(a) Normalized energy



(b) Normalized execution times

Figure 8: Results on Applications

with each of the individual applications themselves (best case savings 11%), because no matter what scheme is used, the time interval between accesses to the device is reduced, thus presenting fewer opportunities to exploit the idle times. Particularly, scanner and mp3 player, which are both regular and have good energy savings individually, show significant drops in energy savings when run together (74% and 55% vs. 11%). When the applications use different devices, as in the last case–viewer and scanner applications, the energy savings is better than the viewer application, but worse than scanner (0% and 74% vs. 24%). Again, this follows intuition, as the viewer contributes little to the overall energy *savings*, but adds up to the overall energy *consumed*.

5.3 Overheads

Microbenchmarks indicate that the annotation-based approach takes about $0.67\mu s$ (the cost of a system call) to obtain the context, whereas with the stack-based approach it is $5\mu s$. [2]

Our algorithm is linear over the number of processes (p) and the number of devices (d), and is of complexity $O(p \cdot d)$. This computation is performed every time a device-related system call is executed. The resulting overhead of the implementation is negligible. To illustrate, the MP3 player application causes a total overhead of about 10ms *due to the algorithm* when the device is suspended on 11 occasions. Further, in all experiments, the device I/O variations are found to be much greater than algorithm overhead, so increases in execution times with contextual timeouts are primarily due to device power mode switches. This is also apparent from the fact that in many cases, the execution times of contextual and conventional timeouts are the same.

6 Related Work

Software controlled energy management, beyond timeout based schemes [5] is explored in multiple recent efforts (e.g., [3] and [25]). [22] recognizes the high fraction of energy consumed by the network interface in handheld devices and recommends aggressive energy management of devices to offer power savings while minimizing user-visible latency. A currency-based system for managing, accounting for, and distributing energy is discussed in [30].

[2] Measured for a program with 10 stack frames. Note that the cost of the stack-based approach depends on the depth of the call stack, and increases by a few instructions for each stack frame.

Efforts to manage the energy consumption of peripherals have been undertaken both (1) with the involvement of the applications using them, as well as (2) at lower layers and transparent to applications. Examples of (1) include [4], where a profile-based approach is used to study the tradeoff between an application's fidelity (based on the quality of the data), and system power consumption. This is then used, along with the history of the application's behavior, to dynamically adapt according to the desired battery lifetime. A similar approach is used in [14], where the power consumption of an application, for a fixed input and environment, is predicted using its history. In [11], applications inform the operating system about their device usage through a set of API calls, and this information is used to control the power modes of devices. In [1], the operating system exposes to applicatins the power modes of devices. Adaptive applications (i.e., those that can use one device instead of another as needed) can exploit this information for their power-aware device usage. In [27], I/O system calls made by the applications are marked deferred/abortable, so as to allow such calls to be batched together, thus extending low-power idle periods. The main impediment in all these approaches is their reliance on changes to applications' sources. This limits their general utility and additional burdens on programmers. In contrast, our approach does not require programmers to be aware of energy management at all, and it works seamlessly with existing programs.

Efforts in (2) include [2], where service requests reaching a device are modelled as finite-state markov chains and then used to predict future accesses. Similarly, [20] uses renewal theory to model service requests to a wireless LAN device, assuming a pareto distribution of user input. Since these schemes are oblivious to the context of the executing application, their predictability is limited by the accuracy of the models used. In CTs, the context is obtained at the application level itself.

Our approach relies on predictions to passively manage peripherals (i.e., without causing any changes to the other parts of the system). There have been many efforts where subparts of the system are actively managed for reduced energy. In [9] and [10], power management of an 802.11 wireless device is carried out by redesigning the communication protocol between the device and the access point. [27] actively batches device service requests to increase low power idle periods. In [7], the compiler is responsible for transforming the application so as to batch requests, whereas this is accomplished by modifying the operating system's scheduling policy in [15]. In [23], a predetermined task schedule in a real-time task is used to generate an energy-efficient schedule for the I/O devices such that task deadlines are not missed. CTs are independent of and makes no assumptions about the schedule used, but are unsuitable for hard real-time systems. [21] and [8] propose using support ASICs and other VLSI design techniques to manage system peripherals so that the main processor can be switched to a low power mode during idle periods. The latter uses EWMA to estimate CPU idle times. A problem discussed therein is how to deal with a short idle period following a long one, that is, how to deal with an overestimation of the idle period. The problem is solved by saturating the estimated idle-time using a saturation constant. In contrast, in this paper, frequent mispredictions are assumed to be caused by irregular applications. Irregularity is addressed by reverting to conventional timeouts as necessary.

7 Conclusions and Future Work

This paper introduces the notion of contextual timeouts (CTs) for aggressive energy savings in the peripherals of handheld devices. An implementation of CTs addresses both application workloads well-suited to the CT approach and workloads better addressed with conventional timeout techniques. Experimental evaluations utilize a range of workloads representing typical usages of handheld or portable devices. With these workloads, CT is compared against both an established energy savings technique and the ideal case of an oracular technique. Evaluations also utilize actual workloads of real world application, again demonstrating the utility of the CT approach to peripheral power management.

Not addressed by our research is the user-driven management of display subsystems on handheld devices, since such management must take into account user behaviors and desires. A representative UI-based approach of value here is described in [31]. Further, beyond power savings, there are additional effects of frequent power mode transitions in devices, such as device lifetime or longevity. Such reliability issues are an interesting topic for future studies.

Two key steps remain to be undertaken in our future work. The first is to generalize CT's mixed compiler- and system-based approach to reducing the power consumption and managing MANET (Mobile Area NETwork) systems. That is, we wish to extend the per-platform solutions presented in this paper to the

distributed and mobile systems domain. The concrete scenarios studied in our group are from the robotics domain, considering emergency rescue scenarios with distributed sensors and autonomous robots [17]. The second step involves combining CT with an orthogonal DVS/DFS technique to achieve a system-wide overall energy management service. Toward this end, we have already successfully integrated CTs with the device windows approach for aggressive device management described in [16].

References

[1] Manish Anand, Edmund Nightingale, and Jason Flinn. Ghosts in the machine: interfaces for better power management. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, 2004.

[2] E. Chung, L. Benini, A. Bogliolo, Y. Lu, and G. De Micheli. Dynamic power management for nonstationary service requests. *IEEE Transactions on Computers*, 51(11):1345–1361, 2002.

[3] Carla Ellis. The case for higher-level power management. In *HOTOS-7*, 1999.

[4] Jason Flinn and M. Satyanarayanan. Managing battery lifetime with energy-aware adaptation. *ACM Transactions on Computer Systems*, 22(2):137–179, 2004.

[5] Richard Golding, Peter Bosch, and John Wilkes. Idleness is not sloth. Technical report, HP Labs, 1996.

[6] Steven D. Gribble. Uc berkeley home ip http traces, 1997. http://www.acm.org/sigcomm/ITA/.

[7] T. Heath, E. Pinheiro, J. Hom, U. Kremer, and R. Bianchini. Code transformations for energy-efficient device management. *IEEE Transactions on Computers*, 53(8):974–987, 2004.

[8] Chi-Hong Hwang and Allen Wu. A predictive system shutdown method for energy saving of event-driven computation. *ACM TODAES*, 5(2):226–241, 2000.

[9] Robin Kravets and P. Krishnan. Application-driven power management for mobile communication. *Wireless Networks*, 6(4), 2000.

[10] Robin Kravets, Karsten Schwan, and Ken Calvert. Power-aware communication for mobile computers. In *Proceedings of MoMuC-6*, 1999.

[11] Yung-Hsiang Lu, Luca Benini, and Giovanni De Micheli. Power-aware operating systems for interactive systems. *IEEE Transactions on VLSI Systems*, 10(2):119–134, 2002.

[12] Giovanni De Micheli and Luca Benini. System level power optimization: techniques and tools. *ACM Trans. Design Automation of Electronic Systems*, 5(2):115–192, 2000.

[13] Ibm microdrive. http://www.hitachigst.com/tech/techlib.nsf/products/Microdrive_1GB.

[14] Dhushyanth Narayanan, Jason Flinn, and M. Satyanarayanan. Using history to improve mobile application adaptation. In *Proceedings of WMCSA-3*, 2000.

[15] Ripal Nathuji and Karsten Schwan. Reducing system level power consumption for mobile and embedded platforms. In *Proceedings of ARCS*, 2005.

[16] Ripal Nathuji, Balasubramanian Seshasayee, and Karsten Schwan. Combining compiler and operating system support for energy efficient i/o on embedded platforms. In *Proceedings of SCOPES*, 2005.

[17] Keith O'Hara and Tucker Balch. Distributed path planning for robots in dynamic environments using a pervasive embedded network. In *AAMAS*, 2004.

[18] Vern Paxson and Sally Floyd. Wide-area traffic: The failure of poisson modeling. *ACM SIGCOMM*, 24(4):257–268, 1994.

[19] M. Rajagopalan, S. Debray, M. Hiltunen, and R. Schlichting. Cassyopia: Compiler assisted system optimization. In *HOTOS*, 2003.

[20] T. Simunic, H. Vikalo, P. Glynn, and G. De Micheli. Energy efficient design of portable wireless systems. In *Proceedings of ISLPED*, 2000.

[21] Mani Srivastava, Anantha Chandrakasan, and Robert Brodersen. Predictive system shutdown and other architectural techniques for energy efficient programmable computation. *IEEE Transactions on VLSI Systems*, 4(1):42–55, 1996.

[22] Mark Stemm, Paul Gauthier, Daishi Harada, and Randy Katz. Reducing power consumption of network interfaces for hand-held devices. In *Proceedings of MoMuC-3*, 1996.

[23] V. Swaminathan and K. Chakrabarty. Energy-conscious, deterministic i/o device scheduling in hard real-time systems. *IEEE Trans. on Computer-Aided Design of Integrated Circuits & Systems*, 22:847–858, 2003.

[24] Sanjay Udani and Jonathan Smith. The power broker: Intelligent power management for mobile computing. Technical report, University of Pennsylvania, 1996.

[25] Amin Vahdat, Alvin Lebeck, and Carla Ellis. Every joule is precious: the case for revisiting operating system design for energy efficiency. In *Proceedings of the 9th workshop on ACM SIGOPS European workshop*, 2000.

[26] M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. In *Proceedings of USENIX Symposium on OSDI*, 1994.

[27] Andreas Weissel, Bjorn Beutel, and Frank Bellosa. Cooperative i/o: A novel i/o semantics for energy-aware applications. In *Proceedings of OSDI-5*, 2002.

[28] Lucent orinoco silver wlan card. http://www.orinocowireless.com.

[29] P. Young. Recursive estimation and time-series analysis, 1984. Springer-Verlag.

[30] Heng Zeng, Carla Ellis, Alvin Lebeck, and Amin Vahdat. Ecosystem: managing energy as a first class operating system resource. *ACM SIGPLAN Notices*, 37(10):123–132, 2002.

[31] L. Zhong and N. K. Jha. Energy efficiency of handheld computer interfaces: Limits, characterization and practice. In *MobiSys*, 2005.