A Programming Environment with Runtime Energy Characterization for Energy-Aware Applications

Changjiu Xian
Department of Computer Science
Purdue University
West Lafayette, Indiana
cjx@cs.purdue.edu

Yung-Hsiang Lu
School of Electrical and Computer Engineering
Purdue University
West Lafayette, Indiana
yunglu@purdue.edu

Zhiyuan Li
Department of Computer Science
Purdue University
West Lafayette, Indiana
li@cs.purdue.edu

ABSTRACT

System-level power management has been studied extensively. For further energy reduction, the collaboration from user applications becomes critical. This paper presents a programming environment to ease the construction of energy-aware applications. We observe that energy-aware programs may identify different ways (called *options*) to achieve the desired functionalities and choose the most energy-efficient option at runtime. Our framework provides a programming interface to obtain the estimated energy consumption for choosing a particular option. The energy is estimated based on runtime energy characterization that records a set of runtime conditions correlated with the energy consumption of the options. We provide the procedure and general guidelines for using the environment to construct energy-aware programs. The prototype demonstrates that (a) energy-aware applications can be programmed easily with our interface, (b) accurate estimates are achieved by integrating multiple runtime conditions, and (c) the framework can facilitate the collaboration among multiple devices to obtain significant energy savings (15% to 41%) with negligible time and energy overhead (<0.35%).

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: design studies

General Terms

Design, Performance

Keywords

energy-aware application, programming environment, energy characterization

1. INTRODUCTION

Reducing energy consumption is important for today's computer systems. Many energy conservation techniques have been proposed from hardware level to operating system (OS) level [2]. For over a decade, power management studies have been focusing on accurately predicting the idleness in workloads such that hardware components can be shut down or slowed down to save energy. These studies have been performed extensively and additional improvements become difficult. For further significant energy reduction, the collaboration from user applications becomes critical. Since the idleness in workloads is ultimately determined by applications, they can adjust their workloads to create more opportunities for power management. Such applications are called *energy-aware applications*. For example, a file-download program in a battery-operated computer can download compressed files and then decompress them. This consumes less energy than downloading the uncompressed files directly if the compression ratio is high and the wireless channel is congested. For another example, web browsers often cache data in a local disk. When the data are requested later, they can be retrieved from the local disk if it is spinning. If the disk is sleeping, however, retrieving the data from the server may be faster and consume less energy than spinning up the disk. These options may be specific to individual applications; according to the end-to-end argument in system design [7], such functionalities should not be placed into operating systems.

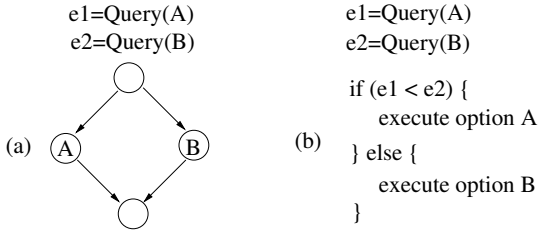

Figure 1: (a) Energy consumption of different execution paths. (b) Pseudo-code of choosing options based on estimated energy consumption.

Existing studies [1, 5, 6, 10] are specific to individual programs and difficult to generalize. In this paper, we present a general programming environment in which energy-aware programs can be constructed more easily. An energy-aware program may identify different ways (called *options*) to achieve desired functionalities and choose the most energy efficient option. The energy consumption of different options may vary, due to different runtime conditions (e.g., the power states of the hardware). Our framework provides an application programming interface (API) for obtaining energy estimates of the options at runtime to decide which option to choose. Figure 1 illustrates how this API works. Suppose the program two choices: A and B. For the first example described earlier, we may have option A to download the uncompressed files and option B to download the compressed files and then decompresses

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ISLPED'07, August 27–29, 2007, Portland, Oregon, USA.
Copyright 2007 ACM 978-1-59593-709-4/07/0008 ...$5.00.

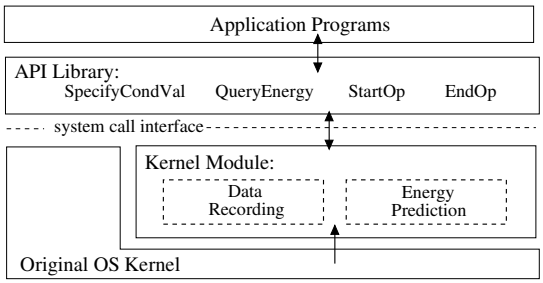

Figure 2: The architecture of the energy-aware programming environment. The arrows show the information flow between different entities.

them. For the second example, we have option A to retrieve the data from the local disk and option B to retrieve from the remote server. In order to decide which option to choose, the program uses our API to obtain the energy estimates for both options, as shown in Figure 1 (a). The pseudo-code in Figure 1 (b) shows that the option consuming less energy will be chosen.

In order to estimate the energy of the options, we record a set of runtime conditions (the data size, the remote IP to connect, etc.) that correlate with the energy consumption of the options as they execute. This is called *energy characterization*. We perform it at runtime to adapt to new cases, for example, a new IP is repeatedly used. The runtime conditions are used to estimate energy consumption of the future instances of the options. Our framework allows integration of both application-level conditions (e.g., the IP to connect) and OS-level conditions (e.g., the power state) for accurate energy estimation. The framework is general because it does not require understanding the specific meanings of the options inside applications. The experimental results based on a prototype demonstrate that (a) energy-aware applications can be programmed easily with our interface, (b) accurate estimates are achieved by integrating multiple conditions, and (c) the framework can facilitate the collaboration among multiple devices to obtain significant energy savings (15% to 41%) with negligible time and energy overhead (<0.35%).

2. RELATED WORK

Several studies have been conducted to save energy at the application level. Flinn et al. [5] experiment with reducing the quality of service, such as the frame size and the resolution for multimedia applications, when the battery energy goes below a certain level. Weissel et al. [8] cluster IO requests to create long idle periods for power management. Rong et al. [6] propose migrating processing to a server to save the client's power. Zhong et al. [10] present several guidelines in designing graphical user interfaces with lower energy consumption. Anand et al. [1] allow programs to choose an already-on component (for example, Bluetooth or WiFi) to obtain data with lower energy consumption. These studies are specific to individual programs and difficult to generalize. In contrast, we provide a general programming environment to ease the construction of energy-aware programs.

Previous studies have performed online prediction of execution time and power consumption of tasks. Dinda [4] presents a system to predict the running time of a computation-bound task. The time is computed from linear time series predictions of host load. Curtis-Maury et al. [3] predict the power and performance of tasks using hardware events. Our prediction method has several major differences from these studies. First, existing methods are for CPU-bounded tasks while ours is also suitable for I/O intensive tasks.

```
SpecifyCondVal("op1", NET, RECV, "IP", ip);
e1 = QueryEnergy("op1");
   ...
e2 = QueryEnergy("op2");

if(e1 < e2) {
    StartOp("op1");
    execute op1
    EndOp("op1");
} else {
    StartOp("op2");
    execute op2
    EndOp("op2");
}
```

Figure 3: The code sample for an energy-aware program with two options.

Second, previous studies consider only a single runtime condition while we consider multiple runtime conditions. Third, they focus on OS level information while we integrate application knowledge with OS information for accurate estimation.

3. ENERGY-AWARE PROGRAMMING ENVIRONMENT

3.1 Overview

Our framework allows applications and OS to collaborate for energy reduction. The design exploits the application knowledge of programmers and reduces their burden on energy estimation. We define the following responsibilities for the programmers: providing the OS with the values of application-level runtime conditions, querying the energy estimates of the options to decide which to choose, marking the start and the end of each option, and revising the options that do not save energy. The responsibilities of the OS include: measuring the energy consumption of options, obtaining the values of the runtime conditions in OS level, and integrating all runtime conditions to estimate energy consumption. Figure 2 shows an overview of the system design. The framework is an extension of existing OS and implemented as a separate kernel module in order to minimize the modification of the original OS. The framework has two parts: (a) an API library that can be linked with programs and (b) a kernel module in the OS to support the functionalities defined by the API. The kernel module has two submodules for recording energy consumption and runtime conditions and predicting the energy consumption of options.

3.2 Application Programming Interfaces

The following API routine is for specifying the application-level runtime conditions.

`SpecifyCondVal(opname, component, mode, cond, val)`

The parameter `opname` is a unique tag and it identifies an option provided by the programmer. The parameter `component` indicates the hardware component (e.g., network card) used by that option. The parameter `mode` specifies the functionality (e.g., `send` or `receive`). The parameter `cond` names an application-level condition predefined in our framework and it will be explained in Section 3.4. The `val` is the value of the condition. The usage of the API routine is illustrated in Figure 3. Suppose option *op1* is to retrieve data from a remote computer. The programmer needs to specify the computer's IP address. This API routine can be called

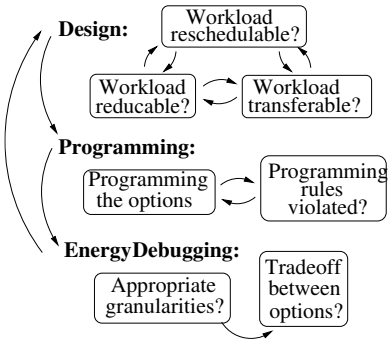

Figure 4: The procedure for constructing energy-aware applications.

multiple times to specify multiple runtime conditions for each option. The following API routine is for querying the energy estimate of an option before it executes.

`QueryEnergy(opname)`

We provide two API routines for marking each option.

`StartOp(opname), EndOp(opname)`

As shown in Figure 3, the routine `StartOp` is inserted before each option in the program to inform the OS that the option will start immediately. The routine`EndOp` is inserted at the end of each option to inform that the option has completed.

3.3 Application Construction

Figure 4 shows the three stages to construct an energy-aware application within our framework. In the design stage, the programmer identifies different options. The options are then implemented by the programmers in the programming stage, making calls to our API routines where they are needed. In the debugging stage, the programmer tests the program and evaluates whether the implemented options actually save energy. Based on the finding, the programmer may revise the original design.

Programmers are responsible for designing the options by following a number of common approaches: (a) Reducing the workload. For example, online map may remove unessential details such as terrain information to reduce data transmission if significant energy can be saved. (b) Rescheduling the workload. For example, a data processing program may cluster its disk reads to create long idleness and the disk can be shut down to save energy. (c) Transferring the workloads between different components. For example, a battery-operated computer may offload a large amount of computation to a server if the communication energy is less than the computation energy. This results in a transfer of workload from the processor to the network card.

In the programming stage, the programmer of the options should follow a number of rules as follows. *Options can be nested*, as illustrated in Figure 5 (a). Within the option "op1", two options "op3" and "op4" are identified. We allow such options by nesting "op3" and "op4" within "op1". *Options can share code*, as illustrated in 5 (b). Both options "op1" and "op2" call function `foo`. *Each option should be complete*. Figure 5 (c) shows a violation of this rule, where a `goto` statement jumps out of the option for error handling and makes the `EndOp` unreachable. Some language supporting exception handling can have similar cases. To satisfy the rule, the programmer can place the error or exception handling code within the option or call `EndOp` first before jumping. *Options should not be interleaving*. There are two general relationships between any

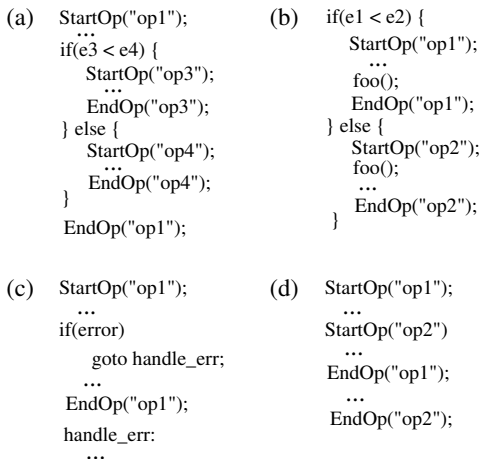

Figure 5: The programming rules: (a) nesting and (b) sharing code are allowed, but (c) jumping out of the option and (d) interleaving options are not allowed.

two options: they are exclusive to each other or one is nested within the other. For two exclusive options, before one option ends, another option should not start. For a nested option, one should be started and completed within the other option. Figure 5 (d) shows that "op1" and "op2" are interleaving and this makes their purpose unclear and is not allowed. Our future work will use compilers to enforce these programming rules for the options.

In the energy debugging stage, the programmer runs the program to examine energy savings. Two pieces of information are provided by our API to the programmer. The first piece shows whether one option always consumes more energy than the other during the testing. If so, then the program can remove the more energy-consuming option. The second piece shows whether the overhead of making the API calls is worth while. To justify such an overhead, the granularity of each option should be large enough such that its energy consumption is greater than the energy consumed by calling our API routines. Based on our prototype and experimental setup (Section 4), the API calls need to execute about 500 C-program statements (e.g., y = x+1;) without accessing IO component. For IO intensive programs, the energy overhead is equivalent to transmitting 110 bytes through a wireless network or reading 660 bytes from the microdrive. Our framework can quantify the energy overhead and the energy consumption of the options.

3.4 Runtime Conditions

As our experimental results in Section 4.3 will show, it is often advantageous to simultaneously consider multiple runtime conditions for energy estimation of options. For example, the estimation for reading local files has an error of 180% if considering only the file size without also considering the power state of the disk and whether the file is cached in memory. When the power state and the cache status are also considered, the error is reduced to be 6%. Table I lists the information currently used in our framework for energy estimation. Each condition can be obtained from either application or OS. The conditions that are easier to obtain using application knowledge are assigned to the application level. For network transmission, the programmer needs to specify the remote IP to connect and the load on the network card. The load is measured by the number of bytes to transmit and it is determined by the program. For example, the option may transmit a whole file, a part of the file, or a buffer of data in the memory. For disk IOs, the programmer needs to specify which file to access and how many bytes to read/write. For the processor, the workload is measured by the

Table 1: Information used for the energy estimation.

information	meaning	level	type
net - receive, send			
load	the number of bytes to transmit	App.	A
IP	the remote IP address	App.	B
BR	the actual bit rate in the near past	OS	A
energy	the energy consumption	OS	
disk - read, write			
load	the number of bytes to read/write	App.	A
file	the file to be read	App.	B
in-mem	whether the file is cached	OS	B
powerstate	the disk's power state	OS	B
energy	the energy consumption	OS	
processor			
load	the needed number of CPU cycles	App.	A
voltage	the processor's voltage setting	OS	A
frequency	the processor's frequency setting	OS	A
energy	the energy consumption	OS	

number of execution cycles and is relatively harder to obtain than the number of bytes. Accurate estimation may be difficult and the program can provide an approximation. For the second example in Section 1, the cycles for decompressing a file can be obtained by profiling decompression on the server. If the program does not specify the cycles, our framework automatically uses the average of the past instances. Our framework achieves better accuracy when more information is available.

The runtime conditions can be classified into two types. Type A: the conditions have numerical values related to the energy consumption directly. A larger value indicates larger energy consumption, such as file size. Type B: the conditions' values are served as only naming to differentiate different cases. Examples include power states and IP addresses. They are regarded as characters instead of numerical values. We need to treat the two types differently in the prediction algorithm as explained in next section.

The number of conditions can be expanded for three reasons. First, our API `SpecifyCondVal` is uniform to different conditions. Second, our prediction algorithm does not require understanding the specific meanings of the conditions. Third, we have defined a common form to represent each condition: a variable to indicate the type, a variable to indicate the level, and a functional pointer to the function used to obtain the value of the condition. These three parameters are provided when a new condition is added to the framework. For an application-level condition, its value is specified by the program so the function pointer can be null.

3.5 Energy Characterization and Prediction

We characterize the energy consumption of the options in programs by recording both the energy and the correlated runtime conditions as the options execute. The energy model in our previous study [9] is used to obtain the energy information. The model uses power parameters of devices and the kernel information about tasks to determine the energy consumption of individual programs in multitasking systems. The accuracy and overhead of our framework are evaluated with actual measurement in Section 4.

We use the recorded data for prediction. However, if a program is newly installed into the system and runs for only a few times, there are insufficient historical data about the program for prediction. We thus allow programs to share data. We record the energy and conditions in five tables, corresponding to CPU, disk read, disk write, network send, and network receive. The columns include a set of runtime conditions and the energy consumption. Each row corresponds to one execution instance of the option. To limit the

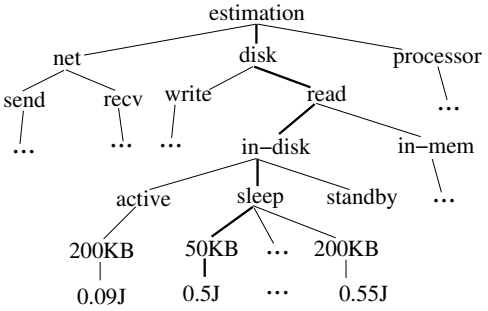

Figure 6: Our energy estimation is based on a decision tree and can handle incomplete information.

memory consumption, each table has only 1000 rows. Five tables use about 100 KB memory and is 0.3% of a PDA with 32MB memory. When the table is full, the oldest row is overwritten. We record the runtime conditions when `StartOp` is called and record the energy when `EndOp` is called. Energy prediction is performed when `QueryEnergy` is called. The input of our prediction algorithm includes the historical data in the five tables and the current values of the runtime conditions in both application level and OS level.

Our prediction algorithm can be viewed as a decision tree shown in Figure 6. Each leaf corresponds to one row of the table. The estimation is a traversal of the tree. One example is to estimate the energy for reading data from a local disk, where the current runtime conditions are: the data are not in memory, the disk is sleeping, and the bytes to read is 50KB. We traverse the tree through the path: disk → read → in-disk → sleep → 50KB → 0.5J. However, this is an ideal case when the current values of the conditions have exact match in the table. When there is no exact much for the conditions' values, the traversal will stop at an internal node of the tree and cannot determine which child of the node is the next step. For example, suppose the current conditions are {disk, read, in-disk, standby, and 200KB} and there is no row containing "standby" in the table. The traversal will stop at the node "in-disk" and cannot proceed. To handle the incomplete information, we pick every sibling of "standby" as the next step to resume the traversal. Suppose the sibling "sleep" is the next step, the following step is then 200KB. The purpose is to use as much available information as possible for accurate estimation. Since we traverse every sibling of a missing value, multiple leaf nodes may be obtained and we use their average as the estimation.

The above method considers all siblings of a missing value of the Type B condition. Since Type-A has numerical values correlated with energy consumption, we interpolate energy values over the missing value of the condition. This is much more accurate than using the average energy of all sibling values of the condition. Since the relation between the condition and the energy can be either linear (e.g., file size) or non-linear (e.g., CPU frequency). We use piece-wise linear regression to handle both cases.

4. EXPERIMENTS

We have built a prototype of the framework in Linux 2.4.18. In this section, we use the API to construct a concrete application to demonstrate the estimation accuracy, the energy savings, and the time and energy overhead.

4.1 Application Construction

We consider a web browser as the application because it uses all three major components: processor, disk, and network. This is for demonstrating that our framework can help multiple components

collaborate to save energy. The web browser caches data in a local disk. We assume the web contents change slowly and studying dynamically generated web pages are out of the scope of this paper. We use a timeout of 24 hours to expire the local data. Before the timeout expires, there are three options for the data present in both local and remote locations: *op1*- fetching the data from the local disk, *op2*- retrieving the data from the remote server, and *op3*- retrieving the remote data in compressed form and then decompressing them using the local processor. We choose the option with the least energy estimate at runtime.

The programming is similar to the code shown in Figure 3. For option *op1*, the program specifies which file to read and the number of bytes to read. The number of bytes is equal to the size of the file in uncompressed form. For option *op2*, we need to specify from which server (IP) the file will be downloaded and the number of bytes to receive. The number of bytes is also equal to the size of the uncompressed file. For option *op3*, the number of bytes to receive is equal to the size of the file in compressed form. Since *op3* needs to decompress the file, the execution cycles needed by the CPU should be specified. The web server needs to support the following functions: (a) Profiling the compression and decompression for the files to obtain the compression ratios and the cycles for decompression. Our client needs to profile the correspondence between the cycles given by the server the actual cycles in the client. (b) Providing either the compressed or the uncompressed version upon request. We currently have both compressed and uncompressed versions available in the server. This needs more storage space but it eliminates the delay of compression. The evaluation of the on-the-fly compression is omitted in this paper due to space limit.

4.2 Experimental Setup

Our experiments are performed in the Integrated Development Platform (IDP) by Accelent Systems running Linux 2.4.18. The IDP uses Intel PXA250 as the processor and provides probing points to measure the power consumption of different hardware components. We install an Orinoco wireless network card and an IBM Microdrive. To measure the actual energy consumption to validate our framework, we use another computer with a data acquisition card (DAQ) from National Instruments. The processor, microdrive, and the wireless card on IDP are connected to the DAQ for energy measurement.

We use 200 files for the experiments. The file sizes range from 1 KB to 300 KB, as the typical file sizes in web cache. The files have various types, such as web pages, Word documents, Powerpoint slides, images, etc. We compress and decompress the files using `gzip`. The compression ratio (the compressed file's size divided by the original file's size) ranges from 6% to 78% and the decompression time on IDP ranges from about 100 to 550 milliseconds.

4.3 Estimation Errors

The errors of energy estimation is defined as $\sqrt{\sum_{i=1}^{n}((e_i' - e_i)/e_i)^2}$, here n is the number of estimates, e_i is the measured energy consumption, and e_i' is the estimated consumption. Figure 7 (a) shows the measured energy for reading local files in different cases: the files are cached in memory, the files are not cached and the microdrive is in active, standby, or sleeping state. Figure 7 (b) shows the errors of energy estimation using different types of information. If we know only the energy of the past instances of file reading, we compute the average energy of the past instances as the estimation. This results in error of 221%. If the file size is also known, a linear regression of energy over file size is used for the estimation. However, the error is not significantly reduced because the energy is still highly varied even for the same file size. If we know whether the file is cached in the memory, the error is reduced to 98%. Finally, if we know the power state of the microdrive, the error is greatly reduced to about 6%.

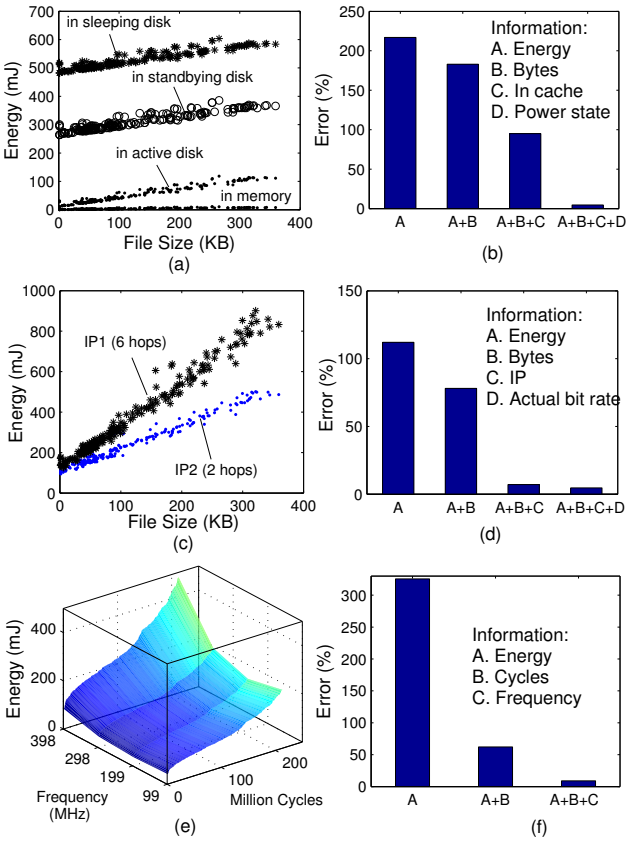

Figure 7: The system's energy consumption and estimation errors for (a)(b) reading local files, (c)(d) fetching remote files, and (e)(f) decompression.

Figure 7 (c) shows the measured energy of fetching files from remote servers and Figure 7 (d) shows the estimation error from using different types of information. The error is about 115% with only energy information and 82% with file size information. If the remote IP is known, the error is reduced to be about 9%.

Figure 7 (e) shows the measured energy for decompressing the files and Figure 7 (f) shows that the estimation error error is 326% with only energy information, 68% with cycles, and 7% with CPU frequencies.

4.4 Energy Savings and Overheads

The energy-aware application chooses different options at runtime to save energy. We compare with three methods that uses only one option: *A*- always fetching files from the local disk, *B*- always fetching files in uncompressed form from the remote server, *C*- always fetching compressed files from the remote server and decompressing the files locally. We name our method *D*- dynamically choosing one of the three options by querying the energy estimates.

Figure 8 (a) shows the energy consumption of the four methods over each file. Method *B* can save more energy than method *A* when the file sizes are small because it avoids the significant wakeup overhead and small files require little transmission energy. The energy savings of method *C* over method *B* is highly dependent

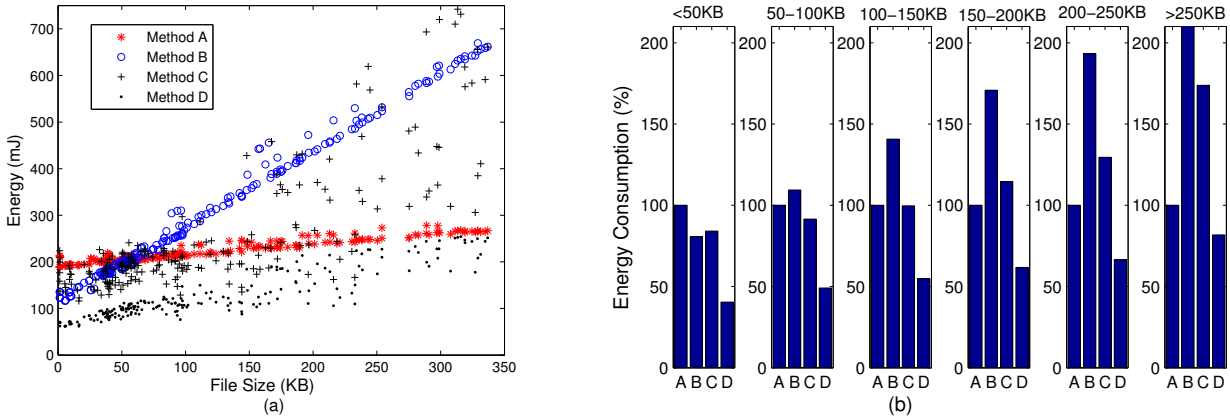

Figure 8: This figure shows the energy consumption of the four methods: *A*, *B*, *C*, and *D* (a) for each file and (b) for different ranges of file sizes.

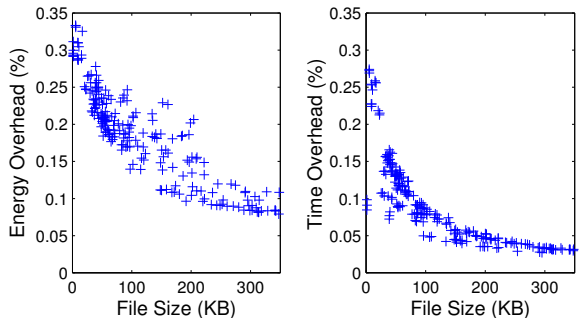

Figure 9: The time and energy overhead of method D.

on individual files that have greatly varied compression ratios and decompression times. Method *D* always chooses the option with the lowest energy estimate at runtime and thus achieves the lowest energy consumption.

Figure 8 (b) shows the average energy consumption of the four methods in different ranges of file sizes. The energy is normalized to method *A* as 100%. The energy savings of method *D* in each range is 41% (<50KB), 40% (50KB-100KB), 39% (100KB-150KB), 35% (150KB-200KB), 22% (200KB-250KB), and 15% (>250KB).

The measured time overhead of executing all API in the program is about 350 microseconds. The energy overhead is about 0.21 mJ and it includes the energy consumption by the whole system. Figure 9 (a) and (b) shows the total time and energy overhead incurred by our API with respected to the total energy consumption of method *D*. Both overheads are less than 0.35%.

5. CONCLUSION

This paper presents a general programing environment to integrate application knowledge and OS information for effective energy reduction. The prototype in Linux demonstrates that (a) energy-aware applications can be programmed more easily using our API, (b) accurate estimates are achieved by using multiple runtime conditions, and (c) the framework can facilitate the collaboration among devices for significant energy savings with negligible overhead. Our future work will study more application scenarios, especially the programs that can have a large number of options. The overhead of code size and storage space for such programs will be examined.

6. ACKNOWLEDGMENT

This work is supported in part by National Science Foundation CNS-0347466 and CCF-0541267. Any opinions, findings, and conclusions or recommendations are those of the authors and do not necessarily reflect the views of the sponsors.

7. REFERENCES

[1] M. Anand, E. B. Nightingale, and J. Flinn. Ghosts in The Machine: Interfaces for Better Power Management. In *International Conference on Mobile Systems, Applications, and Services*, pages 23–35, 2004.

[2] L. Benini and G. D. Micheli. System-Level Power Optimization: Techniques and Tools. *ACM Transactions on Design Automation of Electronic Systems*, 5(2):115–192, April 2000.

[3] M. Curtis-Maury, J. Dzierwa, C. D. Antonopoulos, and D. S. Nikolopoulos. Online Power-Performance Adaptation of Multithreaded Programs using Hardware Event-Based Prediction. In *International Conference on Supercomputing*, pages 157–166, 2006.

[4] P. A. Dinda. Online Prediction of the Running Time of Tasks. In *SIGMETRICS/Performance*, pages 336–337, 2001.

[5] J. Flinn and M. Satyanarayanan. Energy-Aware Adaptation for Mobile Applications. In *ACM Symposium on Operating Systems Principles*, pages 48–63, 1999.

[6] P. Rong and M. Pedram. Extending The Lifetime of A Network of Battery-powered Mobile Devices by Remote Processing: A Markovian Decision-based Approach. In *Design Automation Conference*, pages 906–911, 2003.

[7] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-To-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[8] A. Weissel, B. Beutel, and F. Bellosa. Cooperative IO- A Novel IO Semantics for Energy-Aware Applications. In *Operating Systems Design and Implementation*, pages 117–129, 2002.

[9] C. Xian and Y.-H. Lu. Energy Reduction by Workload Adaptation in a Multi-Process Environment. In *Design Automation and Test in Europe*, pages 514–519, 2006.

[10] L. Zhong and N. K. Jha. Graphical User Interface Energy Characterization for Handheld Computers. In *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pages 232–242, 2003.