Embedded Operating System Energy Analysis and Macro-modeling

T. K. Tan[†], A. Raghunathan[‡], and N. K. Jha[†]
† Dept. of Electrical Eng., Princeton University, NJ 08544
‡ NEC, C&C Research Labs, Princeton, NJ 08540

Abstract

A large and increasing number of modern embedded systems are subject to tight power/energy constraints. It has been demonstrated that the operating system (OS) can have a significant impact on the energy efficiency of the embedded system. Hence, analysis of the energy effects of the OS is of great importance. Conventional approaches to energy analysis of the OS (and embedded software, in general) require the application software to be completely developed and integrated with the system software, and that either measurement on a hardware prototype or detailed simulation of the entire system be performed. Since this process requires significant design effort, unfortunately, it is typically too late in the design cycle to perform high-level or architectural optimizations on the embedded software, restricting the scope of power savings.

Our work recognizes the need to provide embedded software designers with feedback about the effect of different OS services on energy consumption early in the design cycle. As a first step in that direction, this paper presents a systematic methodology to perform energy analysis and macro-modeling of an embedded OS. Our energy macro-models provide software architects and developers with an intuitive model for the OS energy effects, since they directly associate energy consumption with OS services and primitives that are visible to the application software. Our methodology consists of (i) an analysis stage, where we identify a set of energy components, called energy characteristics*, which are useful to the designer in making OS-related design trade-offs, and (ii) a subsequent macro-modeling stage, where we collect data for the identified energy components and automatically derive macro-models for them. We validate our methodology by deriving energy macro-models for two state-of-the-art embedded OS's, μC/OS and Linux OS.*

Acknowledgments: This work was supported by DARPA under contract no. DAAB07-00-C-L516.

1 Introduction

Embedded operating systems form a critical part of a wide range of complex embedded systems, and provide benefits ranging from hardware abstraction and resource management to real-time behavior. The energy effects of the OS have great bearing on the energy efficiency of the overall embedded system. In the past, most of the embedded OS-related investigations have centered around performance issues. In recent years, OS-related issues concerning energy consumption of embedded software have been studied. The problem can be broken down into two aspects. First, the energy consumption overhead of the OS needs to be studied. This work is generally called OS energy characterization [1, 4]. Second, the effects of using different OS's or different usages of an OS need to be investigated too. Initial investigations of this problem have been done in [1, 4, 8, 10, 17].

In this paper, we discuss the motivation for performing OS energy characterization and propose a methodology to do it systematically. Our focus is software oriented, with emphasis on the methodology for performing OS energy characterization. It comprises two parts. The first part is *analysis*, which is concerned with identifying a set of *energy components*, called *energy characteristics*. The second part is *macro-modeling*, which is concerned with obtaining macro-models for the energy characteristics. It involves the process of experiment design, data collection, and macro-model fitting. As far as we know, this work is the first attempt to tackle energy macro-modeling of an embedded OS. To demonstrate our approach, we present experimental results for two well-known OS's, namely, μC/OS [7] and Linux OS.

1.1 Related Work

High-level software energy reduction strategies are usually concerned with the OS. Some general ideas on adapting software to manage energy consumption were presented in [5, 6, 9]. Vahdat *et al.* [17] proposed revisiting several aspects of the OS for potential improvement in energy efficiency. Lu *et al.* [10] proposed and implemented OS-

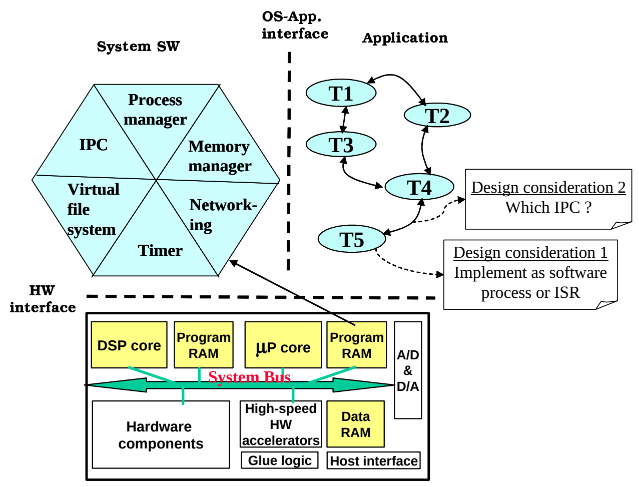


Figure 1. A typical embedded system

directed power management in Linux and showed that it can save more than 50% power compared to traditional hardware-centric shut-down techniques. Bellosa [2] presented thread-specific online analysis of energy-usage patterns that are fed back to the scheduler to control the CPU clock speed.

Performance analysis of the OS has been investigated in the past [12, 18]. In addition, other work has been done to analyze the energy consumption of the OS. Energy analysis of a real-time OS was first performed by Dick *et al.* [4], who developed an energy simulation and analysis framework for μC/OS-SPARClite based embedded systems. Cignetti *et al.* [3] modeled the power consumption of the $PalmOS^{TM}$ family of devices based on discrete device states, and provided an energy simulator based on this power model. Li *et al.* [8] presented issues concerning the implementation of a low power OS and demonstrated the energy advantage of the event-driven TinyOS over a general purpose OS (eCos). Acquaviva *et al.* [1] characterized the energy consumption of eCos, focusing in particular on the relationship between energy consumption and processor frequency.

2 Motivation for OS Energy Characterization

Consider an embedded system consisting of many components, one of which is a microprocessor that hosts all the software tasks. The software tasks use an OS as the run-time engine. A conceptual diagram for such a system is shown in Fig. 1. Assume that the software for this embedded system has the task configuration shown in the figure. *Task T1* through *task T4* deal with background processing such as managing the memory, controlling the LCD, etc. *Task T5* looks out for new data coming from the A/D converter, performs some preprocessing, and passes the data to *task T4*. Given this multi-task specification, there are several ways to actually implement the system. In particular, for *task T5*, we have two options:

C1: In this case, *task T5* is implemented as an actual software process. After performing preprocessing, it passes the data to *task T4* through inter-process communication (IPC).

C2: In this case, *task T5* is implemented as an interrupt service routine (ISR) or signal handler for *task T4*. The ISR or signal handler is activated periodically using a hardware timer. Since *tasks T4* and *T5* share the same process space, passing data from *task T5* to *task T4* does not incur any IPC overhead.

Designers acquainted with the underlying implementation of an OS will know that the *C2* implementation has less overhead (delay and energy) compared to *C1*. However, *C1* is a cleaner (in terms of ease of programming) implementation compared to *C2*. Therefore, some trade-off has to be made. Apart from that, choosing the actual mechanism for IPC also requires a trade-off between various objectives. A decision can be made intelligently only if the metric under consideration (the energy overhead) is quantified. One approach for quantifying the energy overhead of *C1* over *C2* is to actually implement both and compare them in a simulation environment. However, this is very time-consuming. If some type of *energy characteristic data* for the chosen OS are made available, the comparison can be made much more efficiently. This is especially so if the energy characteristic data are provided in the form of energy macro-models. Our work in this paper is a first step towards making this possible.

3 Overall Methodology for OS Energy Characterization

Fig. 2 illustrates the overall methodology for OS energy characterization. There are four steps involved, namely:

1. In the energy effect analysis step, we identify the essential components of the embedded OS one should characterize.

2. After identifying the energy components to characterize, we generate directed test programs that isolate these components from each other.

3. In this step, the test program is compiled and linked with the OS, and fed into an energy simulation tool. This step requires a low-level energy simulation framework that is capable of executing the application software together with the OS and reporting

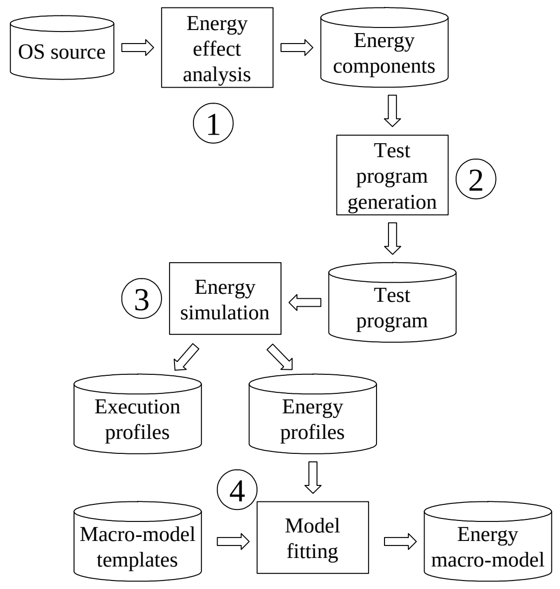


Figure 2. Overall methodology for OS energy characterization

the energy consumption on a function instance basis. Such simulators exist, and were reported in [4, 13].

4. The execution and energy profiles generated from the previous step is subject to further analysis and model fitting to obtain the energy macro-models.

Step 1 is the *energy analysis* stage, whereas Step 2 to Step 4 belong to the *energy macro-modeling* stage. We discuss these two stages of our methodology in the next two sections.

4 OS Energy Analysis

For the purpose of energy analysis, we can view an OS as a multi-entry multi-exit program (MEME-P). Such a view is depicted in Fig. 3, where the big circle is a conceptual representation of the OS-application interface. Dotted lines in the circle represent (conceptually) possible execution paths in the OS. Notice that some paths overlap in some sections, depicting the common fact of program modularity. Some of the entry-exit pairs belong to system call interfaces, whereas others belong to implicit paths of execution within the OS, *i.e.*, they do not directly correspond to system calls in the application code. Examples of implicit execution paths include those triggered by interrupts. In the following discussion, we shall call the entry-exit pairs belonging to the system call interfaces SCEEP (system call entry-exit pair), and the entry-exit pairs belonging to implicit execution paths IEEP (implicit entry-exit pair).

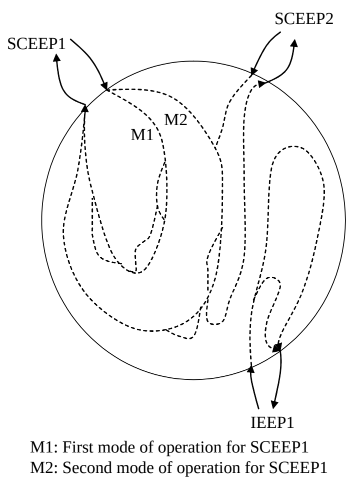

SCEEP1
SCEEP2
M1
M2
IEEP1
M1: First mode of operation for SCEEP1
M2: Second mode of operation for SCEEP1

Figure 3. An MEME-P view of an OS

In a typical OS, a SCEEP can be overloaded with a few groups of paths because the system function it represents is overloaded with many modes of operations. For example, the `read()` system function in the Linux OS can be used to read from a file, from the network, or from a terminal. These different modes of operations correspond to very different groups of paths for the SCEEP of the `read()` function. Similarly, IEEP could also be overloaded in the sense that different groups of paths may be traversed, depending on the state of the OS.

Guided by the MEME-P view, the objective of OS energy analysis is to identify a useful set of SCEEP's and IEEP's, and to classify the paths between them into groups amenable to macro-modeling. The energies consumed while traversing these paths are the *energy characteristics* of the OS. We broadly classify the *energy characteristic data* into two categories, namely the *explicit group* and the *implicit group*. We discuss each group separately in the following sub-sections.

4.1 Explicit Energy Characteristics

By *explicit*, we mean the energy data that is directly related to the OS primitives (system functions). Using the MEME-P view, it basically relates to the energy consumed while traversing the paths between SCEEP's. An OS typically comprises many SCEEP's. A comprehensive set of explicit energy characteristic data should, in principle, cover all the SCEEP's. However, in most practical applications, a selected set of the SCEEP's should suffice in providing useful data for the designers. For system functions that are overloaded with multiple modes of operation, each mode of operation should have its own energy macro-model, even though all the modes of operations share the same SCEEP.

Making energy macro-models available for system functions allows the designers to choose among possible alternatives (system functions) efficiently. Therefore, the significance of these energy macro-models lies not in their ability to provide absolute estimates for the actual energy consumption, but in their ability to facilitate comparison among different alternatives.

4.2 Implicit Energy Characteristics

In contrast to *explicit* energy, *implicit* energy characteristics are not directly related to any OS primitive. That is, the energy is not incurred when exercising an OS primitive, but comes as a result of running the OS engine. Using the MEME-P view, this energy basically relates to the energy consumed while traversing the paths between IEEP's. Similar to the SCEEP, there are usually a few groups of paths for a single IEEP, each of them requiring a separate energy macro-model. The energy consumption along the different groups of paths between the IEEP is characteristic of an OS, and they are called *implicit energy characteristics*. Some of them are:

Timer interrupt energy: This is the energy overhead incurred by the timer interrupt tied to the scheduler.

Scheduling energy: This is the energy overhead of performing rescheduling in the preemptive scheduler.

Context switch energy: This is the energy overhead incurred when a context switch occurs. Note that a call to the scheduler may not always result in a context switch, hence, we separately characterize the context switch and scheduling energy.

Signal handling energy: A signal depicts an OS emulation of a low-level interrupt. Since it is commonly used, its energy overhead should be characterized.

Making energy macro-models available for the above energy components allows the designers to efficiently compute the relative energy cost of different software architectures.

5 OS Energy Macro-modeling

We adopt a *white-box analysis and black-box measurement* philosophy for OS energy macro-modeling. *White-box analysis* refers to the fact that we identify and analyze the energy components of interest by studying the internal operation of the OS. *Black-box measurement* refers to the fact that we measure the energy components by devising experiments that isolate the OS energy of interest by only instrumenting the application code. Black-box measurement and macro-modeling enable the energy models to be used without any knowledge of the OS's internal implementation.

In the following sub-sections, we first discuss step 2 (see Fig. 2) of our methodology for both *explicit* and *implicit* energy characteristics. After that, steps 3 and 4 of our methodology are illustrated.

5.1 Approach to Measure Explicit Energy Characteristics

The energy consumption of an OS primitive (system function) can be characterized by repeatedly calling it in a test program. Care must be taken to extract the "pure" energy cost of the system function, isolating it from the *implicit* energy. For example, in measuring the cost of a message queue *read*, one must make sure that the energy data collected does not include context switch energy. This is achieved by arranging to have both *read* and *write* parts of the experiment in the same software process. Moreover, spurious data resulting from timer interrupt, rescheduling and pre-emptive context switch must be isolated and eliminated. For example, to avoid pre-emptive context switch, the test program must not execute for more than $200ms$ in the case of Linux OS.

Using this approach, we present some characterization results for some commonly used OS primitives in Section 6.

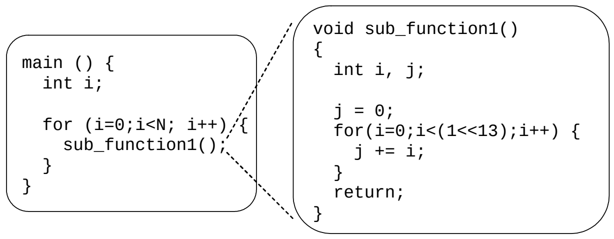
5.2 Approach to Measure Implicit Energy Characteristics

As described in Section 4.2, there are a few key parameters one should measure to obtain the *implicit energy characteristics*. Measuring these parameters is not as straightforward as for *explicit* energy characteristics. We discuss them one-by-one in the following sub-sections.

5.2.1 Context Switch Energy

Context switch energy is a good measure of the chosen OS's ability to perform multi-tasking energy-efficiently. Due to its importance, we present two different approaches to obtain values for this key parameter. Whether both approaches result in similar energy values is an indicator of their accuracy.

The first approach is to arrange two separate experiments, A and B. Experiment A consists of two tasks, connected together by two separate IPC channels going in opposite directions, with a single byte repeatedly being passed back and forth through the two IPC channels. Since two tasks are involved in the activity (reading and writing of the



```
main () {
  int i;

  for (i=0;i<N; i++) {
    sub_function1();
  }
}
```

```
void sub_function1()
{
  int i, j;

  j = 0;
  for(i=0;i<(1<<13);i++) {
    j += i;
  }
  return;
}
```

Figure 4. A test program for revealing context switch energy

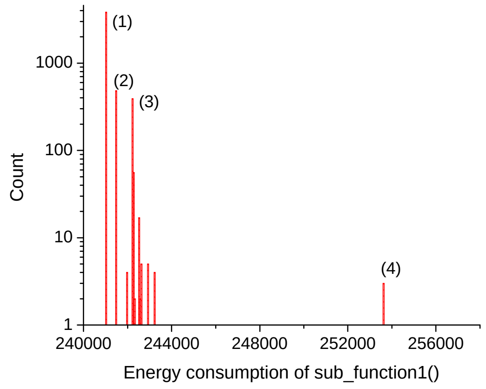


Figure 5. Histogram for the energy consumption of `sub_function1()`

IPC channels), context switches occur repeatedly. Experiment B consists of a similar setup. However, in this case, both the IPC channels are entirely in a single task. As reading and writing of the IPC channels occur, no context switch is involved. By comparing the differences in the energy consumptions between these two experiments, the context switch energy can be isolated.

The second approach is quite different from the first. In this case, the test program consists of a function `sub_function1()` that does nothing, expecting to be preempted by the OS. This test program is shown in Fig. 4. Function `sub_function1()` is called a large number of times by the main test program, and is pre-empted by the OS during some of these calls. Plotting the histogram for the energy consumption of `sub_function1()` reveals the energy incurred by the timer interrupt, rescheduling, and context switch. Such a plot is shown in Fig. 5. Knowing the underlying scheduling mechanism of the OS, we can deduce the origin of each energy cluster in Fig. 5[1] (denoted by (1) – (4) in the figure). The first cluster from the left, *i.e.*, cluster (1), is the nominal energy consumption of

[1] This is exactly the reason we describe our approach as *white-box analysis and black-box measurement.*

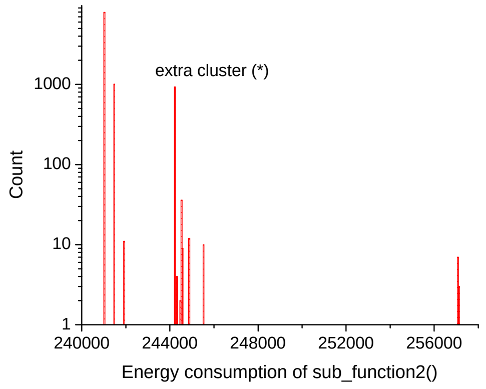


Figure 6. Histogram for the energy consumption of `sub_function2()`

`sub_function1()`. Cluster (2) is due to the timer interrupt, which arrives with a period of $5ms$. Every $10ms$, rescheduling occurs, which results in energy cluster (3). This cluster shows significant dispersion, pointing to the fact that the rescheduling algorithm used in the OS is dynamic. Cluster (4), which has a low count, is attributed to calls to `sub_function1()` during which an actual context switch occurs. To extract the context switch energy from the histogram, we only have to calculate the difference between the fourth and the first energy clusters. As we will show in Section 6, this value turns out to be very close to the value obtained using the first approach.

5.2.2 Timer Interrupt and Rescheduling Energy

Knowing that the second cluster should be attributed to the timer interrupt, we obtain the timer interrupt energy to be the difference between the second cluster and first. Similarly, the scheduling energy is the energy difference between the third cluster and first.

5.2.3 Signal Handling Energy

Signal handling energy needs to be extracted using another experiment. In this experiment, we reuse `sub_function1()` described in Section 5.2.1, naming it `sub_function2()` in this experiment. However, the main test program also sets up an alarm that generates a signal periodically. An alarm signal handler that does nothing (just return) is also established in the main test program. While `sub_function2()` is called repeatedly by the main test program, some invocations will be interrupted by the alarm. Fig. 6 shows the energy histogram for `sub_function2()` in the experiment. Knowing that this setup is the same as the setup in Section 5.2.1 except for signal handling, we attribute the extra cluster to signal handling.

5.3 Energy Simulation and Macro-model Fitting

After devising the test programs required to extract the energy characteristics, we use a low-level energy simulation framework to collect the data. Such low-level energy simulation frameworks exist [4, 13], and are able to report energy consumption of a software function on an instance-by-instance basis.

Macro-model fitting is the last step in our methodology. We use a regression technique developed in [14]. Basically, for the energy characteristics of interest, we first collect a series of data E_1, E_2, ..., E_n using the low-level energy simulator. At the same time, we also choose a macro-model template $E = c_1R_1 + c_2R_2 + \cdots + c_pR_p$ for this energy characteristics. R_j's are the macro-model parameters and c_j's are the unknown coefficients to be determined. Given the collection of energy data, we form a matrix equation using the macro-model template:

$$\begin{pmatrix} R_{1,1} & R_{1,2} & .. & R_{1,p} \\ R_{2,1} & R_{2,2} & .. & R_{2,p} \\ .. & .. & \ddots & .. \\ R_{n,1} & R_{n,2} & .. & R_{n,p} \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ .. \\ c_p \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ .. \\ E_n \end{pmatrix}$$

Solution of this matrix equation using the pseudo-inverse method yields the values for c_j's such that the macro-model best fits the energy data [11].

We next illustrate this macro-model fitting step using an example. Consider the case of building an energy macro-model for the Linux system function `msgsnd()`. Knowing the functionality of this system function, it is straightforward to propose an energy macro-model template expressed as $E = c_1 + c_2x$, where x is the number of bytes processed by this system function, and c_1 and c_2 are the unknown coefficients to be determined. Running the test program generated in the previous step yields a series of energy data E_1, E_2, ..., E_n, corresponding to a series of x's, x_1, x_2, ..., x_n. The corresponding matrix equation used to solve for the unknown coefficients c_1 and c_2 is given by:

$$\begin{pmatrix} 1 & x_1 \\ 1 & x_2 \\ .. & .. \\ 1 & x_n \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \end{pmatrix} = \begin{pmatrix} E_1 \\ E_2 \\ .. \\ E_n \end{pmatrix} \qquad (1)$$

Since it is regression based, the macro-model we propose will have some fitting error with respect to the lower-level energy values it is based on. Evaluation of the error is important to justify the applicability of the macro-models. We use the following relative error metric:

$$\epsilon = \sqrt{\sum_i^n \frac{((\hat{E}_i - E_i)/E_i)^2}{n}} \qquad (2)$$

where E_i's are the energy data values and $\hat{E}_i$'s are the values given by the macro-model. This error metric is an unbiased measure of the statistical significance of the macro-models if the sample size n is much larger than the number of parameters p.

6 Experimental Results

To validate our methodology, this section presents experimental results for energy characterization of two well-known OS's, namely, μC/OS [7] and Linux OS (arm-linux v2.2.2). We use energy simulators [4, 13] for embedded system platforms based on the Fujitsu's SPARClite processor, and the Intel StrongARM processor, to collect low-level energy data for μC/OS and Linux OS, respectively. Discussions of the use and accuracy of such simulators can be found in [4, 13].

Table 1 shows the implicit energy characteristic data for both μC/OS and Linux OS. Since the energy data for μC/OS were derived for a SPARClite-based platform, whereas the energy data for Linux OS were derived for a StrongARM-based platform, the energy characteristic data for μC/OS were also normalized to the StrongARM-based platform so that a meaningful comparison can be made with the Linux OS energy characteristic data. Since two approaches were used to determine the context switch energy, we report for both μC/OS and Linux two context switch energy values, indicated as (1) and (2). The two context switch energy values derived using the two approaches turn out to be very close to each other, thereby providing an empirical validation of the approaches. Note that timer interrupt and rescheduling are inseparable in μC/OS. Therefore, there is a common entry for both in Table 1. Also, there is no concept of signal handler in μC/OS, therefore the entry on signal handling energy for μC/OS is omitted.

Explicit energy characteristic data for μC/OS are reported in Table 2. Data for Linux OS are reported in Table 3. The tables list the energy macro-models for selected sets of system functions that span various sub-systems. For those system functions overloaded with multiple modes of operations, such as `read()` and `write()` in Linux, separate energy macro-models for each mode of operation are reported.

Even though ϵ is large in some cases, such macro-models can still be very useful in providing relatively accurate energy estimation to help with system-level energy optimization. For example, in the context of choosing a proper Linux OS system function for the IPC between *task T4* and *task*

Table 2. Energy macro-models for selected system functions in μC/OS

Sub-systems	System Functions	Macro-models (nJ)	Normalized (nJ)	Remarks
Process Manager	OSTaskCreate()	$E = 55221$	$E = 673.4$	$\epsilon = 1.9\%$
IPC	OSMboxCreate()	$E = 38747$	$E = 472.5$	$\epsilon = 0.4\%$
IPC	OSMboxPend()	$E = 6717$	$E = 81.9$	$\epsilon = 0.8\%$
IPC	OSMboxPost()	$E = 5745$	$E = 70.1$	$\epsilon = 1.2\%$
IPC	OSSemCreate()	$E = 38777$	$E = 472.9$	$\epsilon = 0.4\%$
IPC	OSSemPend()	$E = 6529.8$	$E = 79.6$	$\epsilon = 0.9\%$
IPC	OSSemPost()	$E = 6124.1$	$E = 74.7$	$\epsilon = 0.9\%$
IPC	OSQCreate()	$E = 48320$	$E = 589.3$	$\epsilon = 0.5\%$
IPC	OSQPend()	$E = 9705.2$	$E = 118.4$	$\epsilon = 1.8\%$
IPC	OSQPost()	$E = 9464.1$	$E = 115.4$	$\epsilon = 1.7\%$
Memory Manager	OSMemCreate(x blocks)	$E = 12650 + 3400x$	$E = 154.3 + 41.5x$	$\epsilon = 1.1\%$
Memory Manager	OSMemGet()	$E = 7172.0$	$E = 87.5$	$\epsilon = 1.4\%$
Memory Manager	OSMemPut()	$E = 6700.0$	$E = 81.7$	$\epsilon = 0.8\%$
Timer Manager	OSTimeGet()	$E = 3053.0$	$E = 37.2$	$\epsilon = 1.2\%$
Timer Manager	OSTimeSet()	$E = 3097.1$	$E = 37.8$	$\epsilon = 1.1\%$

Table 1. Implicit energy characteristic data for μC/OS and Linux OS

μC/OS	Energy Overhead	Normalized
Context Switch(1)	66000 nJ	804 nJ
Context Switch(2)	62500 nJ	762 nJ
Timer Interrupt/Sched.	21000 nJ	256 nJ
Linux OS	Energy Overhead	
Context Switch(1)	12570 nJ	
Context Switch(2)	12500 nJ	
Timer Interrupt	450 nJ	
Scheduling	1200 nJ	
Signal Handling	3200 nJ	

T5 described in Section 2, energy macro-models for all the available IPC system functions come in handy. Knowing the typical amount of data being passed, the designer can calculate the energy consumption of all the possible IPC system functions and choose the one with the lowest energy consumption. Even for system functions which do not have alternatives, energy macro-models give designers a quick way of estimating the energy consumption overhead of the system functions.

7 Limitations

Though we have tried to be as comprehensive as possible in our approach to OS energy characterization, the scope of our work has to be limited in some ways. First of all, we target embedded operating systems that are monolithic [16] and run on a single processor. This group of OS's includes Linux, μC-Linux, μC/OS, eCos, *etc*. Second, although the overall methodology should be applicable across multiple OS's, the details of test program generation is nevertheless implementation-dependent. In our case, we used Linux OS as the target OS and leveraged the approach for μC/OS. Third, the focus of this paper is only on OS energy characterization. Though we have pointed out some possible uses of the *energy characteristic data*, systematically using this information in a system-level software energy reduction framework is a topic for future work.

Our methodology involves the use of low-level energy simulators to provide the required energy data as a basis of macro-modeling. Accuracy of the macro-model fitting was discussed in Section 5.3. The aggregate error as a result of macro-modeling and simulation is discussed in [15].

8 Conclusions

We presented a systematic approach to analyze and macro-model the energy consumption of various components in an embedded OS. In the energy analysis stage, we dissected a typical OS and described an approach to identify the essential components of the embedded OS one should characterize. In the macro-modeling stage, we described the experiments we performed to measure the energy characteristics, and the approach we took to obtain the corresponding macro-models. We validated our methodology on two well-known OS's, μC/OS and Linux OS. Energy macro-models obtained using our approach can be very helpful for guided OS-related design trade-offs.

References

[1] A. Acquaviva, L. Benini, and B. Ricco. Energy characterization of embedded real-time operating systems. In *Proc. Workshop Compilers & Operating Systems for Low Power*, Sept. 2001.

Table 3. Energy macro-models for selected system functions in Linux

Sub-systems	System Functions	Macro-models (nJ)	Remarks
Process Manager	fork()	$E_{parent} = 30540$	$\epsilon = 3.6\%$
Process Manager	fork()	$E_{child} = 4300$	$\epsilon = 0.1\%$
Process Manager	getpid()	$E = 177$	$\epsilon = 0.1\%$
IPC	msgsnd(x bytes)	$E = 4752 + 1.08x$	$\epsilon = 1.4\%$
IPC	msgrcv(x bytes)	$E = 4913 + 1.19x$	$\epsilon = 1.3\%$
IPC	pipe write(x bytes)	$E = 1147 + 1.64x$	$\epsilon = 11.2\%$
IPC	pipe read(x bytes)	$E = 690 + 1.65x$	$\epsilon = 14.4\%$
IPC	signal()	$E = 2739.9$	$\epsilon = 0.1\%$
Timer Functions	gettimeofday()	$E = 495$	$\epsilon = 0.1\%$
Timer Functions	settimeofday()	$E = 595$	$\epsilon = 0.1\%$
Memory Manager	calloc(x blocks, size y)	$E = 287 + 0.986xy$	$\epsilon = 5.1\%$
Memory Manager	malloc()	$E = 123$	$\epsilon = 2.0\%$
Memory Manager	free()	$E = 160$	$\epsilon = 27.5\%$
Memory Manager	memcpy(x bytes)	$E = 110 + 0.98x$	$\epsilon = 4.6\%$
File System	open()	$E = 2351$	$\epsilon = 2.1\%$
File System	close()	$E = 696$	$\epsilon = 0.1\%$
File System	file read(x bytes)	$E = 1209 + 1.06x$	$\epsilon = 16.9\%$
File System	file write(x bytes)	$E = 2430 + 2.75x$	$\epsilon = 15.7\%$

[2] F. Bellosa. The benefits of event-driven energy accounting in power-sensitive systems. In *Proc. ACM SIGOPS European Workshop*, Sept. 2000.

[3] T. L. Cignetti, K. Komarov, and C. S. Ellis. Energy estimation tools for the $Palm^{TM}$. In *Proc. ACM MSWWiM 2000: Modeling, Analysis and Simulation of Wireless and Mobile Systems*, Aug. 2000.

[4] R. P. Dick, G. Lakshminarayana, A. Raghunathan, and N. K. Jha. Power analysis of embedded operating systems. In *Proc. Design Automation Conf.*, pages 312–315, June 2000.

[5] K. I. Farkas, J. Flinn, G. Back, D. Grunwald, and J. M. Anderson. Quantifying the energy consumption of a pocket computer and a Java Virtual Machine. In *Proc. SIGMETRICS*, pages 252–263, June 2000.

[6] J. Flinn and M. Satyanarayanan. Energy-aware adaptation for mobile applications. In *Proc. ACM Symp. Operating System Principles*, pages 48–63, Dec. 1999.

[7] J. J. Labrosse. *MicroC/OS-II: The Real-time Kernel.* R&D Books, Lawrence, KS, 1999.

[8] S. F. Li, R. Sutton, and J. Rabaey. Low power operating system for heterogeneous wirelesss communication systems. In *Proc. Workshop Compilers & Operating Systems for Low Power*, Sept. 2001.

[9] J. R. Lorch and A. J. Smith. Software strategies for portable computer energy management. *IEEE Personal Communications*, 5(3):60–73, June 1998.

[10] Y. H. Lu, L. Benini, and G. D. Micheli. Operating-system directed power reduction. In *Proc. Int. Symp. Low Power Electronics & Design*, pages 37–42, July 2000.

[11] R. H. Myers. *Classical and Modern Regression with Application.* Durbury Press, Belmont, CA, 2nd edition, 1989.

[12] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod. Using SimOS machine simulator to study complex computer systems. *ACM Trans. Modeling & Computer Simulation*, 7(1):78–103, 1997.

[13] T. K. Tan, A. Raghunathan, and N. K. Jha. EMSIM: An energy simulation framework for an embedded operating system. In *Proc. Int. Symp. Circuit & Systems*, pages 464–467, May 2002.

[14] T. K. Tan, A. Raghunathan, G. Lakshminarayana, and N. K. Jha. High-level software energy macro-modeling. In *Proc. Design Automation Conf.*, pages 605–610, June 2001.

[15] T. K. Tan, A. Raghunathan, G. Lakshminarayana, and N. K. Jha. High-level energy macro-modeling of embedded software. *IEEE Trans. Computer-Aided Design*, Sept. 2002.

[16] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems: Design and Implementation.* Prentice Hall, Reading, MA, 2nd edition, 1997.

[17] A. Vahdat, A. Lebeck, and C. S. Ellis. Every Joule is precious: The case for revisiting operating system design for energy efficiency. In *Proc. 9th ACM SIGOPS European Workshop*, Sept. 2000.

[18] K. Weiss, T. Steckstor, and W. Rosenstiel. Performance analysis of a RTOS by emulation of an embedded system. In *Proc. Int. Wkshp. Rapid System Prototyping*, pages 146–151, June 1999.