EMSIM : An Energy Simulation Framework for an Embedded Operating System

T. K. Tan†, A. Raghunathan‡, and N. K. Jha†

† Dept. of Electrical Eng., Princeton University, NJ 08544. Email: {tktan,jha}@ee.princeton.edu

‡ NEC, C&C Research Labs, Princeton, NJ 08540. Email: anand@ccrl.nj.nec.com

Abstract— Modern embedded systems are typically built using an operating system (OS) (frequently an embedded OS). As energy consumption has become an important issue in the design of embedded systems (*e.g.*, mobile computing and wireless communication devices), various techniques have been developed for the design of energy-efficient embedded software. In OS-driven embedded systems, the OS has a significant impact on the system's energy consumption directly (energy consumption associated with the execution of the OS functions and services), as well as indirectly (interaction of the OS with the application software).

As a first step towards designing energy-efficient OS-based embedded systems, it is important to develop methodologies to analyze the energy consumption of the embedded OS. We present, in this work, an energy simulation framework that can be used to analyze the energy consumption characteristics of an embedded system featuring the embedded Linux OS running on the StrongARM processor.

I. Introduction

The complexity of embedded system software and the underlying hardware, tight performance and power budgets, and aggressive time-to-market schedules, usually necessitate the use of sophisticated run-time software support. In embedded systems where a single processor executes many different system tasks (each system task may be further divided into communicating processes), the use of an embedded OS for run-time execution support is quite common. Investigations of performance issues related to the use of an OS can be found in [1–4]. However, on modern microprocessors, where idle or sleep modes are usually exploited by software, performance optimization does not necessarily lead to energy optimization. Not much literature deals with energy consumption issues related to an embedded OS.

In this paper, we present *EMSIM*, an energy simulation framework that enables simulation of an embedded Linux OS on a typical hardware platform based on the Intel StrongARM processor. As described in Section III, such a simulation imposes special requirements in terms of modeling the functionalities of the hardware platform.

The remainder of the paper is organized as follows. In the next section, we present some previous work and highlight our contributions. Section III presents the design and validation of our simulator. Section IV concludes.

II. Related Work and Our Contributions

In this section, we discuss related work and highlight our contributions.

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

A. Related Work

Rosenblum *et al.* [5] presented *SimOS*, a machine simulator that simulates the hardware of MIPS-based multiprocessors in enough detail to boot and run an essentially unmodified version of Silicon Graphics' IRIX OS. Energy analysis of a real-time OS was first performed by Dick *et al.* [6], who enhanced the work of [7] and developed an energy simulation and analysis framework for μC/OS-SPARClite based embedded systems. A similar work by Cignetti *et al.* [8] 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. Flinn *et al.* [9] introduced *PowerScope* as a hardware instrumentation tool to map energy consumption to program structures such as processes and procedures.

B. Paper Contributions

Our work includes the following contributions:

- As Linux gains popularity as a general-purpose OS, its use in embedded systems is also increasing. We have modeled the StrongARM microprocessor and the required peripherals in sufficient detail so as to run Linux OS starting from system initialization to the spawning and execution of multiple application processes. This is significantly different from previous work such as [10] wherein an OS cannot be executed in their simulator. We have also included power models for various components in the modeled system so as to enable energy simulation of the embedded software (system and application).
- As far as we know, this is the first energy simulation framework for an embedded system featuring a StrongARM microprocessor and Linux OS. Our energy modeling and accounting methodology can also be easily applied to other embedded system platforms.

III. Design of the Simulator

To enable an OS such as Linux to run in the simulator, we need to model the various system modules in sufficient detail. Besides modeling for the core components such as instruction decoder, pipeline, caches and memory management unit (MMU), other hardware components essential for the functionality of an OS should also be modeled correctly. In the case of Linux OS, we need to have timers to set the pace for task scheduling. We also need to model *Universal Asynchronous Receiver Transmitters* (UARTs), since Linux can use one of the serial ports as the console output. Most importantly, we need to have an interrupt controller to service the interrupt requests from the timers, UARTs and the internal software interrupts.

In the following sub-sections, we discuss various aspects of the design in detail.

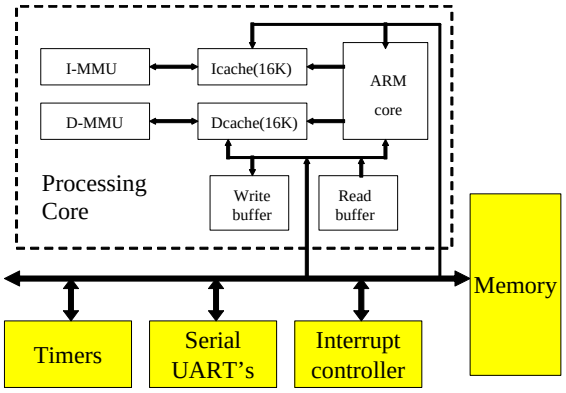

Fig. 1. Modeled embedded system

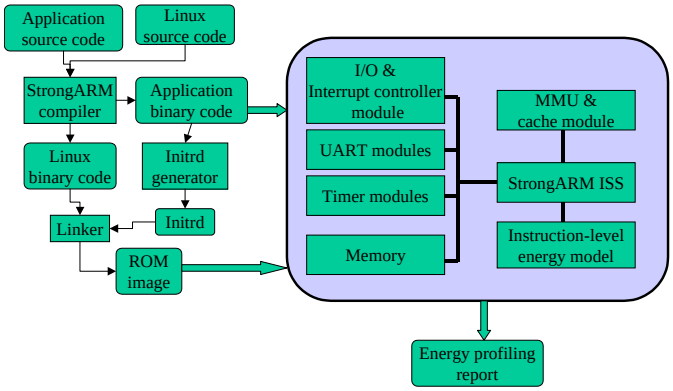

Fig. 2. Energy analysis framework

A. Basic Features

The hardware structure of the system that we model in our simulator is shown in Fig. 1. The corresponding simulation framework is shown in Fig. 2. Shown on the right half of Fig. 2 are the simulation models for all the sub-systems featured in Fig. 1, whereas the sequence of the steps involved in using the simulation framework is shown on the left. As shown in Fig. 2, the simulator includes the following components:

- A simulator for the StrongARM core, consisting of an *instruction-set simulator* (ISS), and simulation models for 16K D-cache and I-cache, and a memory management unit (MMU).
- Simulation model for an interrupt controller.
- Simulation models for two timers.
- Simulation models for two UARTs conforming to the 8250 series. By default, we direct the transmitter of UART0 to the host terminal.
- Simulation model for 32 Mbytes of memory.

We have modeled each component in sufficient detail to enable a version of Linux OS to execute on it. The Linux OS that we adapted for the purpose of simulation has the following features:

- It is *arm-linux* version 2.2.2, configured for the EBSA-110 platform.
- We have configured the *arm-linux* to mount an initial ramdisk (*initrd*) as the root filesystem. The user application code is pre-installed into the *initrd*. At the end of the kernel initialization steps, the user application program is loaded from the ramdisk.
- We have also configured the *arm-linux* to use UART0 as the console. Output from the kernel as well as from the user application can thus be displayed on the host terminal since the transmitter of UART0 is redirected to the host terminal.

B. Usage of the Simulator

The flow diagram depicting the usage of the simulator is also included in Fig. 2. Given the *arm-linux* source code, we generate the *arm-linux* object code using a cross-compiler. In parallel, we also generate the user application object code from the application source code using the cross-compiler. As mentioned in the previous sub-section, we need to pre-install the user application code into the *initrd*. To do that, we use an *initrd* generator. The output of the *initrd* generator is a compressed image of the *initrd*. After that, a linker is used to link the *arm-linux* object code and the compressed *initrd* image into a final ROM image.

At the start, the simulator loads the ROM image into the memory. Notice that the application object code is also loaded by the simulator, as depicted in the flow diagram. This is necessary even though the application is already pre-installed in the *initrd* because we need the function symbol information directly from the application code to enable function energy profiling.

C. Energy Modeling in the Simulator

Simunic et al. [10] have shown that reasonably accurate (within 5% error) power estimation can be obtained even if the power models of the constituent components are inferred directly from the data-sheet information. In our work, we follow this philosophy for most components of the simulator except for the processor, for which we have obtained the StrongARM instruction-level energy models from [11]. The energy consumption of the modeled embedded system in every execution cycle can be expressed as a sum of the energy contribution from the following components:

Processor core: The instruction-level energy models from [11] already include the energy contributions of all parts of the processor core, including cache, MMU, clock generator, timer, interrupt controller, *etc.* Following the original methodology described in [12], the instruction-level energy models are applicable when cache hits occur. From the data provided in [11], the energy consumption of the processor core in each core cycle ranges from 1.17 nJ to 1.68 nJ, depending on the instruction being executed. When cache misses occur, additional energy is consumed as a result of external bus and memory access. We address this next.

Memory: We assume that our memory part is the same as that used in the Western Research Lab's Itsy pocket computer v1.5 [13]. The energy consumption of the memory part in each bus clock cycle can be extracted from the data given in [13]. Note that clock switching [14] is normally enabled by the *arm-linux* kernel, hence the bus clock frequency is half of the core clock frequency. From [13], we estimate the memory energy consumption for each bus clock cycle to be 4.70 nJ.

UARTs: According to [13], the additional power consumption incurred when UARTs are enabled is fairly constant at approximately 44 mW. At a core clock frequency of 206 MHz, we estimate the energy consumption of the UART module in each cycle to be 0.21 nJ.

Other peripherals: Energy models for other peripherals can be added in the future in the same

manner as for the UART.

Our simulator also supports StrongARM's *idle mode*. When the processor is running the Linux kernel without a workload, the idle task in the kernel kicks the processor into *idle mode*. From [13], we obtain the idle mode power consumption of the processor chip to be 65 mW for a clock frequency of 206 MHz. Hence, the idle mode energy consumption is 0.32 nJ per core cycle.

D. Energy Accounting

Correct accounting for the task and function energy is one of the challenging parts of the design of our simulation framework. The energy accounting mechanism of our simulator is task-based. In other words, we keep a separate energy balance sheet for each task running in the simulator. At the beginning of the initialization of the kernel, there is only one task, the *idle-task*. The name *idle-task* might be misleading since this task does a lot of the hardware initialization before spawning another task called *init-task*. The *init-task* continues with kernel software initialization, mounts a root filesystem near the end, and executes the first user-mode program.

The energy balance sheets in the simulator have a one-to-one correspondence with the tasks running in the simulator. At the beginning, there is only *idle-task*. Hence, there is only one energy balance sheet. When the *init-task* is created, a new energy balance sheet is created and the energy value for the new balance sheet is initialized to zero. Anytime a context switch occurs, the simulator is able to detect the context switch non-intrusively and attribute the energy consumption of the embedded system to the running task accordingly. Therefore, at any point in time, the energy value in each energy balance sheet indicates the energy consumption of the embedded system due to the corresponding task. The sum of energy values from all the energy balance sheets equals the total energy consumption of the embedded system.

Energy profiling for the software functions is more involved. Basically, we maintain a *function energy* stack for each task. The function energy stack in the simulator tracks the function call stack for the corresponding task. The only additional information stored in each slot of the function energy stack is the energy value of the task at the entry of a function. When the function exits, the current value of the task energy minus the energy value at the entry is the energy consumption of that particular function instance. The slot is *closed* by storing the energy consumption value in a database.

Since task creation in the Linux OS is always accomplished by task cloning, the function energy stack is also cloned when a task is created. However, the entry energy values for all the slots in the new function energy stack are reset to zero, as is the energy value of the new energy balance sheet. This is necessary to avoid double counting.

The simulated program triggers the end of the simulation by setting the program counter to 0x00000000. Once this happens, the simulator starts consolidating the function energy stacks for all the tasks in the simulator. Basically, all the slots in the function energy stacks are closed and the *partial* energy consumptions of the functions are stored in the energy database. After that, the energy database is stored into a file for post-processing.

E. Simulator Validation

Validation of our simulator consists of two parts. The first part validates the functionality of the simulator, while the second part validates the energy modeling accuracy.

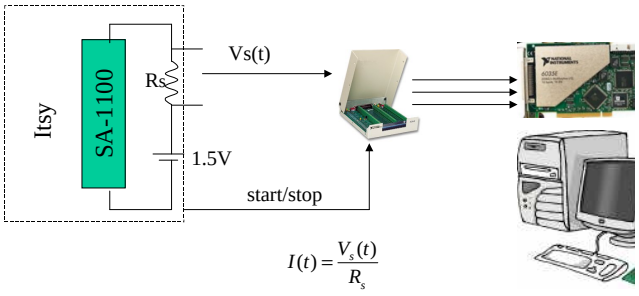

Fig. 3. Energy measurement setup

E.1 Functionality validation

For functionality validation, we ran a few commonly known multi-task programs to make sure that the outputs of the simulations were the same as those obtained from host executions. The multi-task programs we tried include:

- *Dining philosopher:* This program has five tasks to represent five philosophers and another five tasks to represent five forks available. Acquisition of a fork by a philosopher is accomplished through message passing from the philosopher task to the fork task.
- *Priority inversion:* This program demonstrates the priority inversion problem inherent in the Linux OS. It consists of three tasks. It makes calls to the sleep() library function and also tests the *idle-mode* of the simulator.
- *Time-of-flight:* The purpose of this program is to measure the time it takes a message to be transfered across a message queue. The total time includes the time to make the call to *msgsnd()*, the time for the system to transfer the message, the time for context switch, and finally, the time for other end to call *msgrcv()*.

E.2 Energy modeling accuracy

This validation is targeted at energy modeling accuracy of the simulator. We ran a few example programs both in the simulator and on an *Itsy* evaluation board [15]. The *Itsy* board has more components than what we have modeled in the simulator. However, since the *Itsy* board also provides a built-in measurement circuit to allow convenient measurement of the current going from the 1.5V supply to the StrongARM processor chip, we only make comparisons of the processor energy consumption.

We built an energy measurement setup as illustrated in Fig. 3. The measurement setup consists of a computer running *National Instrument*'s data acquisition software called *Labview*, a data acquisition card (DAQ card), and a wire connector box. What we measure directly is the voltage across a sense resistor R_s, which is connected in series with the battery. From the measured voltage V_s, the power supplied to the processor chip is calculated as

$$P_s(t) = V_{dd}\frac{V_s(t)}{R_s} \tag{1}$$

where V_{dd} is its supply voltage. At the beginning of each program, we insert code to set a GPIO pin to HIGH. At the end of the run of the program, we have code to reset the GPIO pin to LOW. We use the GPIO signal $S(t)$ as a gating window for the integration of power $P_s(t)$. The energy consumption of the program is calculated by

$$E(t) = \int_0^\infty S(t)P_s(t)dt \tag{2}$$

TABLE I

DESCRIPTIONS OF THE EXAMPLE PROGRAMS USED IN THE VALIDATION

Program	Description
ins_sort	A program that performs insertion sort on arrays of integers
chksum	A program that performs checksum on arrays of integers
myfrag	A program that implements packet fragmentation in the IP protocol stack
mult	A program that performs matrix multiplication
br_mem	A synthetic program that makes many memory accesses
br_smem	Another synthetic program that makes different memory footprints
edgedet	A program that performs edge detection on a series of images
myqsort	A program that performs quick sort on arrays of integers

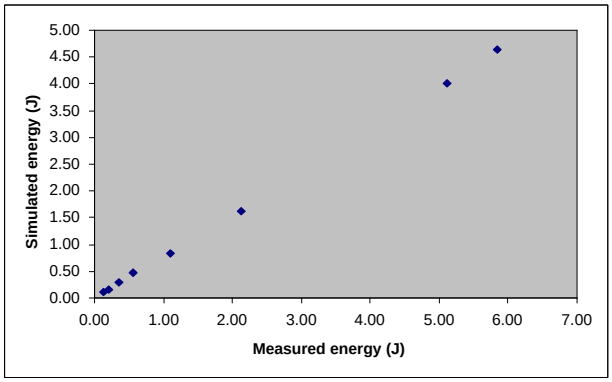

Fig. 4. Comparison between the estimated and measured energy

As shown in Equation (2), the gating signal $S(t)$ ensures that the integration accumulates the power consumption only when the example program is running.

With this measurement setup, we conducted experiments for a set of 9 example programs. Short descriptions of these programs are provided in Table I. The plot in Fig. 4 shows the simulated versus measured energy consumption for these programs. Though the simulated energy does not match the measured energy exactly, a good correlation between the two is observed. This relative accuracy is sufficient to enable software architectural exploration, which is the objective of our future research.

IV. Conclusions and Future Work

In this work, we presented *EMSIM*, an energy simulation framework for an embedded system consisting of a StrongARM microprocessor and the required peripherals. The framework is detailed enough so that we can run the Linux OS starting from initialization to the spawning of multiple application tasks. The energy simulation framework has been validated for both its functionality and modeling accuracy.

This simulator can be used to study and characterize the energy consumption of the Linux OS. Such a work paves the way for our future work in software architectural exploration for low energy consumption.

References

[1] T. Benner, A. Osterling, and R. Ernst, "Comparison of context switching methods for fine grain process scheduling," Tech. Rep. CY-96-1, Institute of Computer Engineering, Technical Univ. of Brauschweig, Germany, 1996.

[2] D. Poirier and K. Jeffay, "An implementation and application of the real-time producer/consumer paradigm," Tech. Rep. 90-038, Department of Computer Science, University of North Carolina at Chapel Hill, 1990.

[3] K. Jeffay, "On latency management in time-shared operating systems," in *Proc. IEEE Workshop Real-time Operating Systems & Software*, May 1994, pp. 86–90.

[4] K. Jeffay and G. Lamastra, "A comparative study of the realization of rate-based computing services in general purpose operating systems," in *Proc. Int. Conf. Real-time Computing Systems & Applications*, Dec. 2000, pp. 81–90.

[5] M. Rosenblum, E. Bugnion, S. Devine, and S. A. Herrod, "Using SimOS machine simulator to study complex computer systems," *ACM Trans. Modeling & Computer Simulation*, vol. 7, no. 1, pp. 78–103, 1997.

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

[7] Y. Li and J. Henkel, "A framework for estimating and minimizing energy dissipation of embedded HW/SW systems," in *Proc. Design Automation Conf.*, June 1998, pp. 576–581.

[8] 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.

[9] J. Flinn and M. Satyanarayanan, "PowerScope: A tool for profiling the energy usage of mobile applications," in *Proc. IEEE Workshop Mobile Computing Systems & Applications*, Feb. 1999, pp. 2–10.

[10] T. Simunic, L. Benini, and G. De Micheli, "Cycle-accurate simulation of energy consumption in embedded systems," in *Proc. Design Automation Conf.*, June 1999, pp. 867–872.

[11] A. Sinha and A. P. Chandrakasan, "JouleTrack - A web based tool for software energy profiling," in *Proc. Design Automation Conf.*, June 2001, pp. 220–225.

[12] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: A first step towards software power minimization," *IEEE Trans. VLSI Systems*, vol. 2, no. 4, pp. 437–445, Dec. 1994.

[13] J. Flinn, K. I. Farkas, and J. Anderson, "Power and energy characterization of Itsy pocket computer (version 1.5)," Tech. Rep. TN-56, Western Research Laboratory, Feb. 2000.

[14] Intel Corporation, *Intel StrongARM SA-1100 Microprocessor Developer's Manual*, Aug. 1999.

[15] W. R. Hamburgen, D. A. Wallach, M. A. Viredaz, L. S. Brakmo, C. A. Waldspurger, J. F. Barlett, T. Mann, and K. I. Farkas, "Itsy: Stretching the bounds of mobile computing," *Computer*, vol. 34, no. 4, pp. 28–36, Apr. 2001.