REFERENCES

[1] R. Kastner and M. Sarrafzadeh. (1999) Labyrinth: A Global Router and Development Tool. [Online]. Available: URL:http://www.cs.ucla.edu/kastner/labyrinth/

[2] F. K. Hwang, D. S. Richards, and P. Winter, "The Steiner tree problem," in *Annals of Discrete Mathematics*. Amsterdam, The Netherlands: North-Holland, 1992, vol. 53, pp. 203–282.

[3] C. Albrecht, "Provably good global routing by a new approximation algorithm for multicommodity flow," in *Proc. ACM/SIGDA Int. Symp. Physical Design*, Apr. 2000, pp. 19–25.

[4] R. C. Carden IV, J. M. Li, and C. k. Cheng, "A global router with a theoretical bound on the optimal solution," *IEEE Trans. Computer-Aided Design*, vol. 15, pp. 208–216, Feb. 1996.

[5] C. Chiang, C. K. Wong, and M. Sarrafzadeh, "A weighted Steiner trees-based global router with simultaneous length and density minimization," *IEEE Trans. Computer-Aided Design*, vol. 13, pp. 1461–1469, Dec. 1994.

[6] X. Hong, T. Xue, E. S. Kuh, C. K. Cheng, and J. Huang, "Performance-driven Steiner tree algorithm for global routing," in *Proc. ACM/IEEE Design Automation Conf.*, 1993, pp. 177–181.

[7] E. Bozorgzadeh, R. Kastner, and M. Sarrafzadeh, "Creating and exploiting flexibility in Steiner trees," in *Proc. ACM/IEEE Design Automation Conf.*, June 2001, pp. 195–198.

[8] R. Kastner, E. Bozorgzadeh, and M. Sarrafzadeh, "Pattern routing: Use and theory for increasing predictability and avoiding coupling," *IEEE Trans. Computer-Aided Design*, vol. 21, pp. 777–791, July 2002.

[9] A. Caldwell, A. Kahng, and I. Markov, "Can recursive bisection alone produce routable placements?," in *Proc. 37th ACM/IEEE Design Automation Conf.*, June 2000, pp. 477–482.

[10] J. Hu and S. Sapatnekar, "A timing-constrained algorithm for simultaneous global routing of multiple nets," in *Proc. ACM/IEEE Int. Conf. Computer-Aided Design*, Nov. 2000, pp. 99–103.

[11] A. Jagannathan, S.-W. Hur, and J. Lillis, "A fast algorithm for context-aware buffer insertion," in *Proc. 37th ACM/IEEE Design Automation Conf.*, June 2000, pp. 368–373.

[12] C. Alpert, "The ISPD98 circuit benchmark suite," in *Proc. Int. Symp. Physical Design*, Apr. 1998, pp. 80–85.

[13] E. F. Moore, *The Shortest Path Through a Maze*, 1959, pt. II, vol. 30, Annals of the Harvard Computation Laboratory.

[14] M. Garey and D. Johnson, "The rectilinear Steiner tree problem is NP-complete," *SIAM J. Appl. Math.*, pp. 826–834.

[15] J. Cong and X. Yuan, "Routing tree construction under fixed buffer locations," in *Proc. 37th ACM/IEEE Design Automation Conf.*, June 2000, pp. 379–384.

[16] J. Ho, G. Vijayan, and C. K. Wong, "A new approach to the rectilinear Steiner tree problem," in *Proc. ACM/IEEE Design Automation Conf.*, June 1989, pp. 161–166.

[17] K. Kozminski, "Benchmarks for layout synthesis – Evolution and current status," in *Proc. ACM/IEEE Design Automation Conf.*, June 1991, pp. 265–270.

[18] M. Lai and D. F. Wong, "Maze routing with buffer insertion and wire sizing," in *Proc. ACM/IEEE Design Automation Conf.*, June 2000, pp. 374–378.

[19] M. Sarrafzadeh and C. K. Wong, *An Introduction to VLSI Physical Design*. New York: McGraw–Hill, 1996.

[20] M. Wang, X. Yang, and M. Sarrafzadeh, "DRAGON: Fast standard-cell placement for large circuits," in *Proc. IEEE Int. Conf. Computer-Aided Design*, Nov. 2000, pp. 260–263.

[21] W. J. Sun and C. Sechen, "Efficient and effective placement for very large circuits," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1993, pp. 170–177.

Analysis of Power Dissipation in Embedded Systems Using Real-Time Operating Systems

Robert P. Dick, Ganesh Lakshminarayana, Anand Raghunathan, and Niraj K. Jha

***Abstract*—The increasing complexity and software content of embedded systems has led to the frequent use of system software to help applications access hardware resources easily and efficiently. In this paper, we present a method for detailed analysis of real-time operating system (RTOS) power consumption. RTOSs form an important component of the system software layer. Despite the widespread use of, and significant role played by, RTOSs in mobile and low-power embedded systems, little is known about their power-consumption effects. This paper presents a method of producing a hierarchical energy-consumption profile for applications as they interact with an RTOS. As a proof-of-concept, we use our infrastructure to produce the power profiles for a commercial RTOS, μC/OS-II, running several applications on an embedded system based on the Fujitsu SPARClite processor. These examples demonstrate that an RTOS can have a significant impact on power consumption. We discuss ways in which application software can be designed to use an RTOS in a power-efficient manner. We believe that this is a first step toward establishing a systematic approach to power optimization of embedded systems containing RTOSs.**

***Index Terms*—Embedded system, energy consumption, low-power, operating system, power consumption, real-time, simulation.**

I. INTRODUCTION

Embedded systems often contain programmable processors and peripherals in addition to application-specific hardware. The complexity of applications and underlying hardware, tight performance and power budgets, as well as aggressive development schedules, require application developers to use runtime support software. This support usually takes the form of a real-time operating system (RTOS), runtime libraries, and device drivers [1]–[7]. RTOSs are used in embedded systems with soft real-time constraints, as well as formal real-time systems with hard real-time constraints. In the interest of brevity, we will use the term RTOS to refer to all operating systems targeting time-constrained embedded systems.

An RTOS provides a number of services to an embedded system designer. It serves as an interface between application software and hardware. For example, the RTOS provides the designer with timer management routines that may be used without detailed knowledge about the timer hardware in the embedded system. In addition to simplifying the use of hardware, the interrupt service routines (ISRs) provided by an RTOS allow hardware to signal an application. The device driver and memory management portions of an RTOS simplify embedded system design by providing the designer with routines to ease the management of hardware resources. An RTOS manages the execution of, and interaction between, tasks in an application. It handles the scheduling of

Manuscript received June 26, 2000; revised April 19, 2002. This work was supported in part by a grant from NEC C&C Research Labs and in part by the George Van Ness Lothrop Fellowship in Engineering. This paper was recommended by Associate Editor Rajesh Gupta.

R. P. Dick is with the Department of Electrical and Computer Engineering, Northwestern University, Evanston, IL 60208 USA (e-mail: dickrp@ee.princeton.edu).

G. Lakshminarayana is with the Alphion Corporation, Eatontown, NJ 07724 USA.

A. Raghunathan is with C&C Research Labs, NEC USA, Princeton, NJ 08540 USA.

N. K. Jha is with the Department of Electrical Engineering, Princeton University, Princeton, NJ 08544 USA.

Digital Object Identifier 10.1109/TCAD.2003.810745

different tasks in an application, ensuring that the highest-priority task has access to an embedded system's hardware resources at any given time. It also provides for communication and synchronization among tasks. In short, it manages the details of task interaction and provides a simplified interface to hardware resources.

Unlike general-purpose operating systems, RTOSs often sacrifice some functionality for the sake of compactness, predictability, and speed. A number of services typically provided by general-purpose operating systems are not useful in most embedded applications, e.g., support for multiple users or complex file-systems. By omitting such features, the size of an RTOS may be reduced, decreasing memory requirements and, therefore, embedded system cost. General-purpose operating systems usually try to complete their duties quickly. However, they typically do not provide a hard guarantee that a task will complete by a certain time. RTOSs differ from general-purpose operating systems by making hard real-time guarantees about the time requirements of the critical services they provide.

Typical applications involve significant use of RTOS primitives, the complex interactions among which are hidden from the application software developer. Although abstracting away the detailed behavior of RTOS services allows embedded system designers to more easily manage complexity, tight performance and power constraints sometimes demand more detailed analysis. The way an RTOS is used has a significant impact on embedded system performance and power consumption. Therefore, designers need to be aware of the impact of RTOS on these design characteristics. Therefore, designers need to be aware of the potential performance and power impact of RTOS use. Commercial RTOS manuals and data sheets typically include estimates of the execution time for various parts of the RTOS for specific hardware configurations. However, vendors do not provide information about RTOS power-consumption characteristics. In addition, state-of-the-art techniques in embedded software power analysis do not clearly separate and analyze power consumed in RTOS components. We propose and demonstrate a method of conducting a detailed hierarchical analysis of the power consumption and execution time of embedded system applications running on a multitasking RTOS. In addition, our work is a first step toward analyzing and characterizing power consumptions of different RTOS components as well as the indirect impact of RTOS usage upon embedded system power consumption.

The rest of the paper is organized as follows. Section II introduces related research and summarizes our contributions. Section III demonstrates the potential impact of an RTOS on embedded system energy consumption, using various illustrative examples. It also describes how insights into RTOS effects on energy can be used to optimize software to reduce energy consumption. Section IV describes our energy analysis infrastructure, and presents an overview of the μC/OS-II RTOS. Section V presents quantitative experimental results on several example embedded software systems, on which we base our analysis of RTOS energy effects. Section VI concludes and makes recommendations to designers of low-power embedded systems that use RTOSs.

II. Related Work and Contributions

The importance of reducing power consumption in embedded systems has now been widely recognized, and a large body of work has focused on estimating, managing, and reducing power consumption in various system components. For hardware design, techniques have been developed to estimate and optimize power consumption starting from the algorithm and architectural design phases, down to the circuit design and technology optimization steps [8]–[12]. Application, semiconductor technology, cost, and time-to-market trends are causing a shift toward increased software content in embedded systems and systems-on-chip. As a result, designers and users of embedded software must be increasingly aware of power issues. While power dissipation is inherently a property of the underlying system hardware, a knowledge of the embedded software that runs on the hardware is useful in order to analyze and improve the system's power-consumption characteristics.

Recognizing the important role played by embedded software in determining system power consumption, researchers have started to investigate techniques for software power analysis and power-efficient software design. Power analysis techniques have been proposed for embedded software based on instruction-level characterization [13], and simulation of the underlying hardware [14]. Techniques to improve the efficiency of software power analysis through statistical profiling have been proposed in [15]. The system-on-chip design paradigm, which enables integration of processors, peripherals, busses, and complex user-defined logic blocks, has fueled research in hardware- and software power-consumption estimation [16]–[21]. Reducing embedded software power consumption through compiler optimizations [22], source-level transformations [18], [23], customized memory management schemes [24], power management schemes [8], [25], device driver and operating system policies [26], and variable-voltage processors [27]–[30] has been investigated. Researchers have also begun investigating methods of using operating systems to dynamically disable peripherals in order to save power [31]. Others have advocated redesigning page allocation and communication policies in order to decrease energy consumption [32].

Our work focuses on understanding and characterizing the power effects of RTOS and application software. Our goal is to provide designers with a method of determining the system-specific changes to the interaction between application software and RTOS that will most effectively reduce system power consumption. The steps required to reduce system power consumption are necessarily dependent on the specific RTOS and processor being used. We applied this method to the μC/OS-II RTOS [3] and applications running on the Fujitsu SPARClite processor. However, our method of hierarchically analyzing RTOS and application software power consumption [33] can be applied to different processors and RTOSs, e.g., an ARM processor running Linux [34]. Others have subsequently used a simulation-based approach to analyze RTOS power consumption [35]. We modeled the SPARClite processor's sleep mode. It was observed that, in applications making heavy use of RTOS services, the RTOS, itself, can consume a significant amount of power. However, in general, the impact of RTOS usage upon application software consumption is more significant than power directly consumed by the RTOS. We present quantitative results for energy and time consumed by different operating system tasks, such as context switching, scheduling, interprocess communication, and timer management. In addition, we present concrete examples of the ways in which information derived from RTOS power analysis can be used to optimize embedded software power consumption. Our method of RTOS power analysis can be used for research on high-level power-modeling of different RTOS components. These models can be incorporated into power-aware system-level design tools.

III. Motivation for RTOS Energy Analysis

In this section, we illustrate, with examples, the impact of RTOS usage on system energy and time consumption. The RTOS energy analysis infrastructure described in Section IV is used to provide a quantitative categorization of the energy and time consumed by different parts of the application and RTOS. Our simulation infrastructure identifies the key sources of energy consumption in the system. Significant savings in energy consumption are obtained by rewriting the application software to use the RTOS in a more energy-efficient manner.

Energy-consumption information is generally more useful than power-consumption information when optimizing an embedded

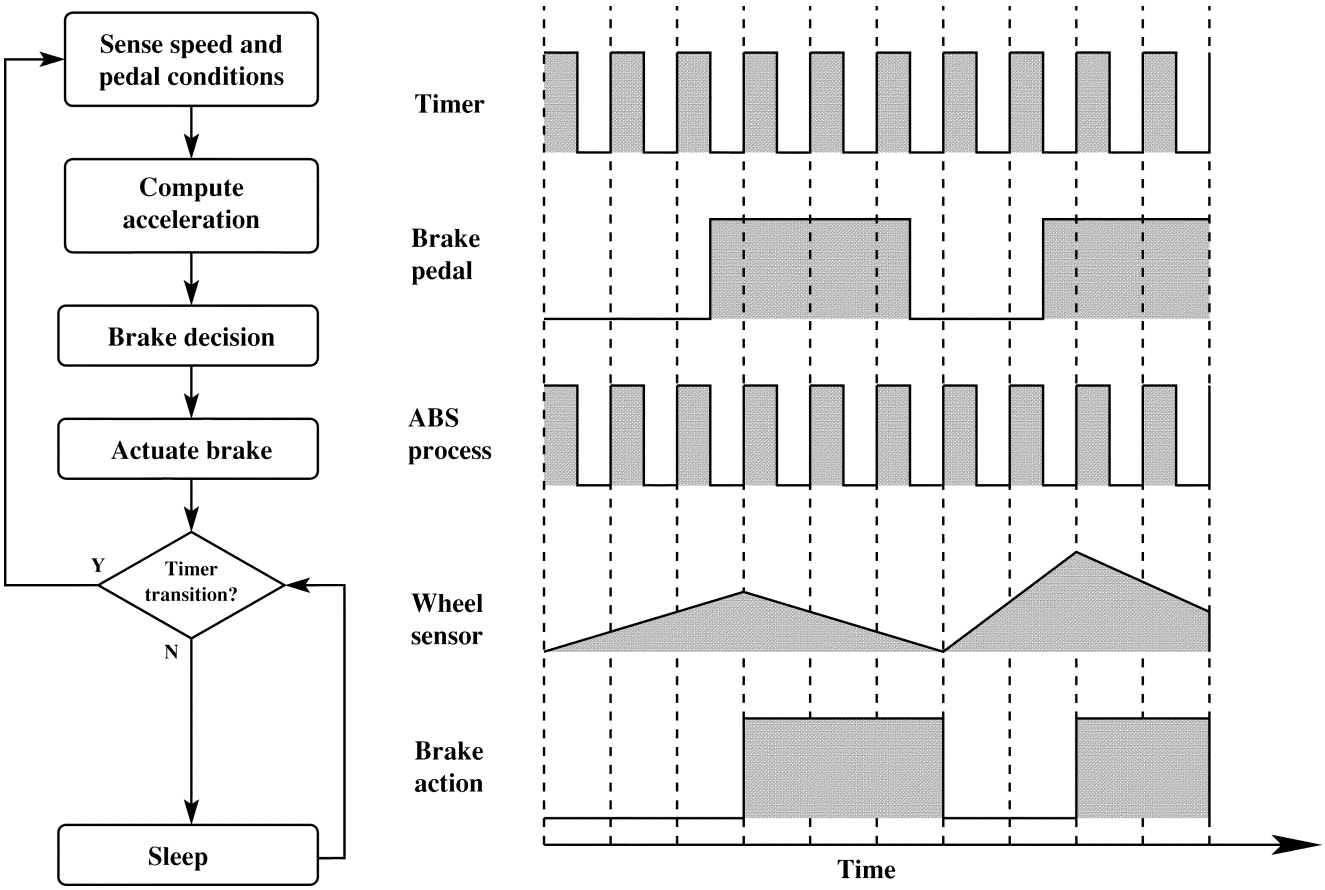


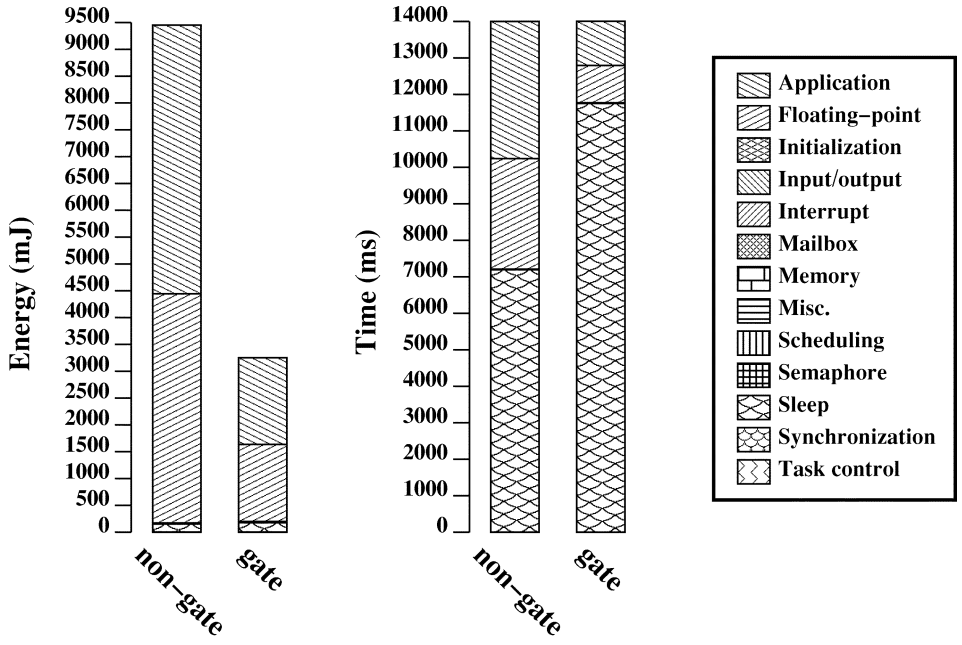
Fig. 1. Straightforward implementation of the ABS example.

system's battery lifespan. Even in situations requiring the optimization of power consumption, e.g., building an embedded system with limited short-term heat dissipation, one may frequently convert an energy-reduced system to a power-reduced system by reducing the system's clock rate, putting it in a reduced power consumption sleep mode part of the time, or reducing the voltage at which some of its components operate. Therefore, we focus on the energy consumption of a number of simulated embedded systems in this paper. In addition, we give time-consumption profiles for these examples. Note that the power consumption profile follows directly from the energy and time consumption profiles.

A. Antilock Braking Example

Our first example is based on embedded software used in an automotive antilock braking system (ABS). The system uses a timer wake-up signal to trigger execution of the ABS process. Consider the flow chart shown in Fig. 1 that shows part of an ABS. The system has been adapted from an example in a design automation manual [36]. In the step marked, sense speed and pedal conditions, the ABS process calls the sense brake pedal and sense speed functions. The sense brake pedal function determines whether the brake is currently depressed. The Sense speed function uses the wheel sensor to determine the current angular velocity (rotation speed) of the wheel. The ABS process then computes the current speed and acceleration of the automobile, and uses the speed, acceleration, and brake pedal status to decide whether to apply the brakes, pump the brakes, release the brakes, or do nothing. This braking decision is conveyed to the actuate brake function that clamps the brake calipers, if appropriate. The simulated vehicle was subjected to an input trace during which its speed and brake pedal conditions change multiple times. The energy consumption profile is shown in the nongate bar of Fig. 2(a).

In the straightforward implementation of the ABS example, illustrated in Fig. 1, the processor is awakened and the ABS process executes with every timer tick. Note that even this straightforward implementation is power-aware: it uses the processor's sleep mode between sensor sampling events instead of continuously leaving the processor in its high-power active mode. However, it frequently executes without changing the condition of the brake calipers. This unnecessary execution requires energy that might otherwise be conserved. By changing the algorithm slightly, such that it only wakes up the processor on a timer tick if the brake pedal is depressed [as shown in Fig. 3], the embedded system's energy consumption is reduced. As shown in the gated energy bar of Fig. 2(a), the energy-optimized implementation of the ABS example consumes 65.0% less energy than the straightforward implementation. Most of the energy savings result from allowing the SPARClite processor to remain in the sleep mode, and the dynamic random access memory (DRAM) to remain in the self-refresh mode, through timer ticks during which it is certain that the brake calipers need not be clamped. As the execution time in each case was 14 s [see Fig. 2(b)], power consumption also reduced by 65.0% in the energy-optimized version. In both versions of this example, operating system and board support services accounted for approximately half of the system's energy consumption. In this example, floating point service routines account for the majority of RTOS and board support energy consumption. Although some of the functions listed in the bar chart's key account for little energy, we have listed all categories to keep the keys of different figures consistent.



(a) (b)

Fig. 2. ABS example: (a) energy and (b) execution time consumption by RTOS service category.

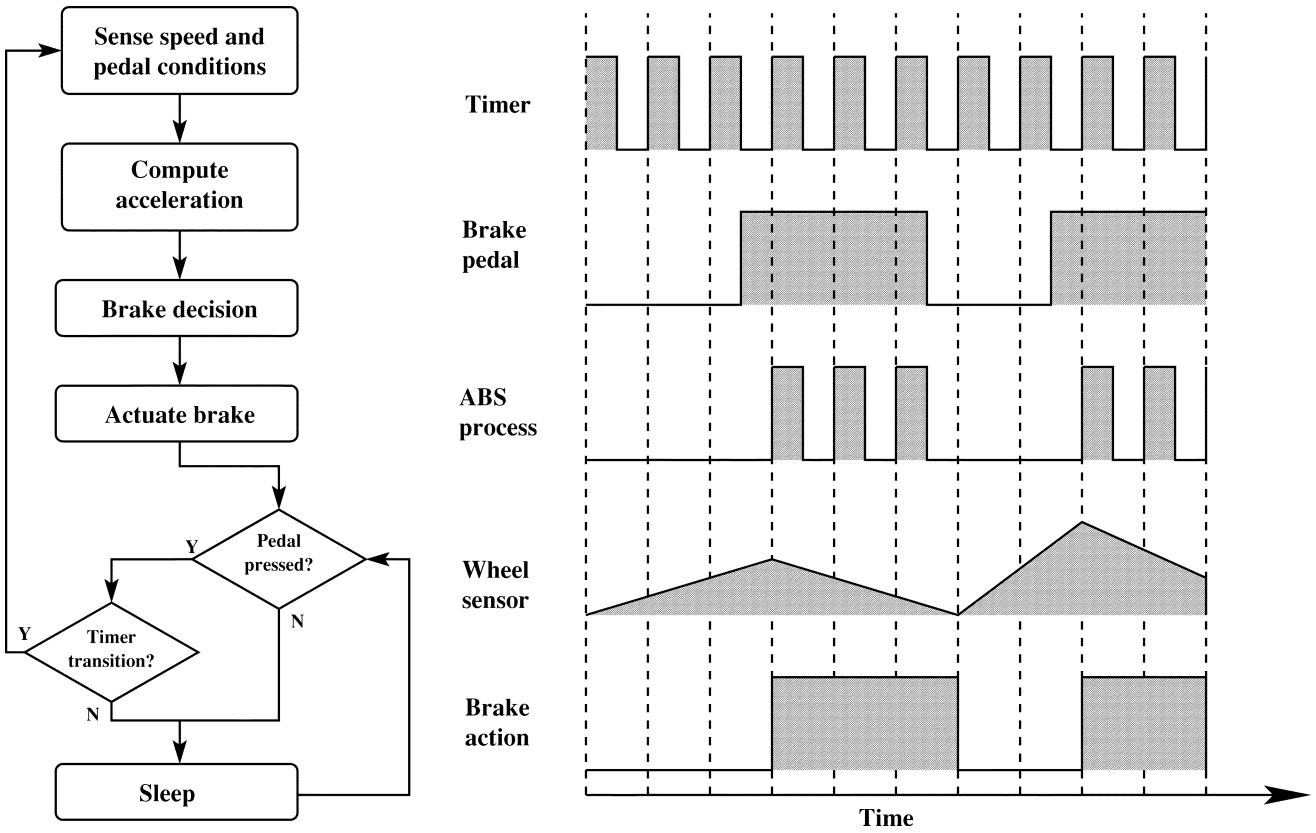


Fig. 3. Energy-optimized implementation of the ABS example.

B. Commodity Trading Agent Example

In our second example, we consider a market composed of commodity trading agents. As shown in Fig. 4, each agent has money and four different types of commodities. These quantities are randomly initialized. Randomly selected agents broadcast, to all other agents, their desire to sell a particular commodity. Agents receiving the broadcast respond with an offer price computed from the agent's supply-price curve for the commodity under consideration. The seller agent uses its supply-price curve to determine whether the highest received offer is higher than its internal valuation of the commodity under consideration at the quantity it currently owns. If so, it sells one unit of the commodity

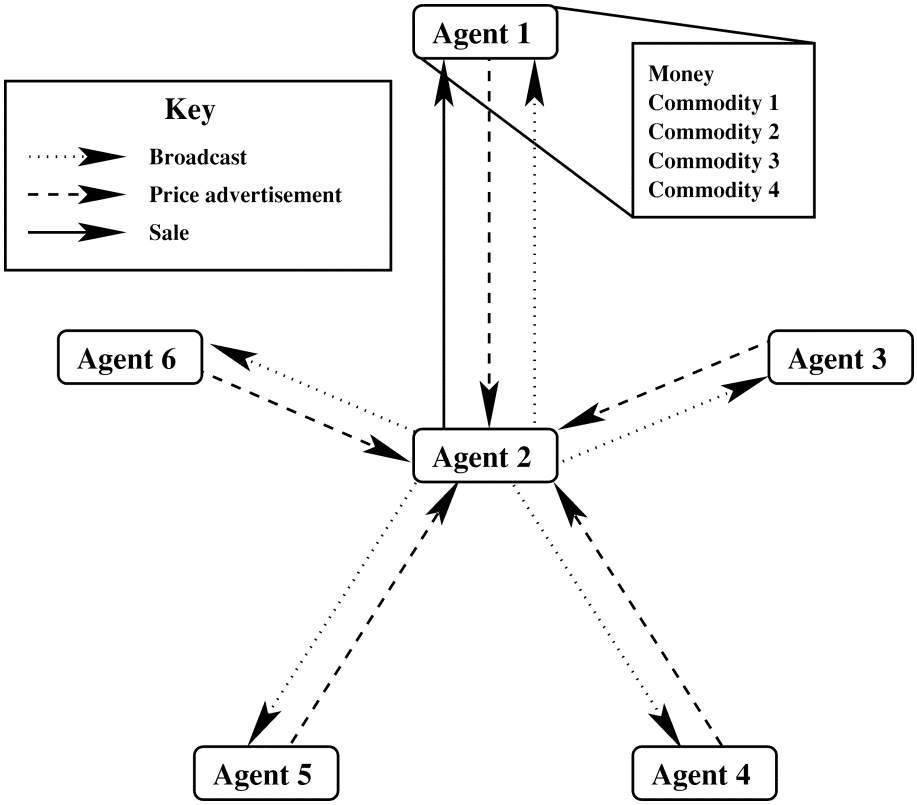


Fig. 4. Overview of the commodity trading agent example.

to the agent making the highest offer. Note that this example is not as conventional as the antilock break and Ethernet examples described in the preceding and following sections. However, this forward-looking example does provide an opportunity to examine the potential impact of hand optimization on application energy consumption.

The mail bar of Fig. 5(a) shows the energy consumption profile for an embedded system running the commodity trading example, when implemented using RTOS mailboxes to transmit messages between agents. In addition, the mail version relies on the RTOS scheduler to manage the activity of different agents. The tuned bar shows the energy consumption for code that is carefully hand-tuned to use shared memory of message communication, and avoid the use of RTOS mailboxes or scheduler. In the mail version, the RTOS is responsible for 95.5% of the embedded system's energy consumption. Interrupt handling, mailbox services, and scheduling, alone, account for 27.6% of the energy consumption. In the tuned version, the RTOS is responsible for 92.2% of the energy consumption. Interrupt handling, mailbox services, and scheduling account for 2.0% of the energy consumption. Note that this example relies very heavily upon RTOS services. In many embedded systems, RTOS energy consumption will account for less than 10% of the total.

As shown in Fig. 5(a), there is an energy cost associated with using the RTOS scheduler and mailboxes to allow a more versatile and maintainable implementation. The tuned version required only 70% of the energy required by the mail version. However, adding new prioritized tasks to the mail version is simple, while changing the behavior of the tuned version is more difficult. In this case, a designer may trade off flexibility and maintainability for energy savings.

C. Ethernet Interface Example

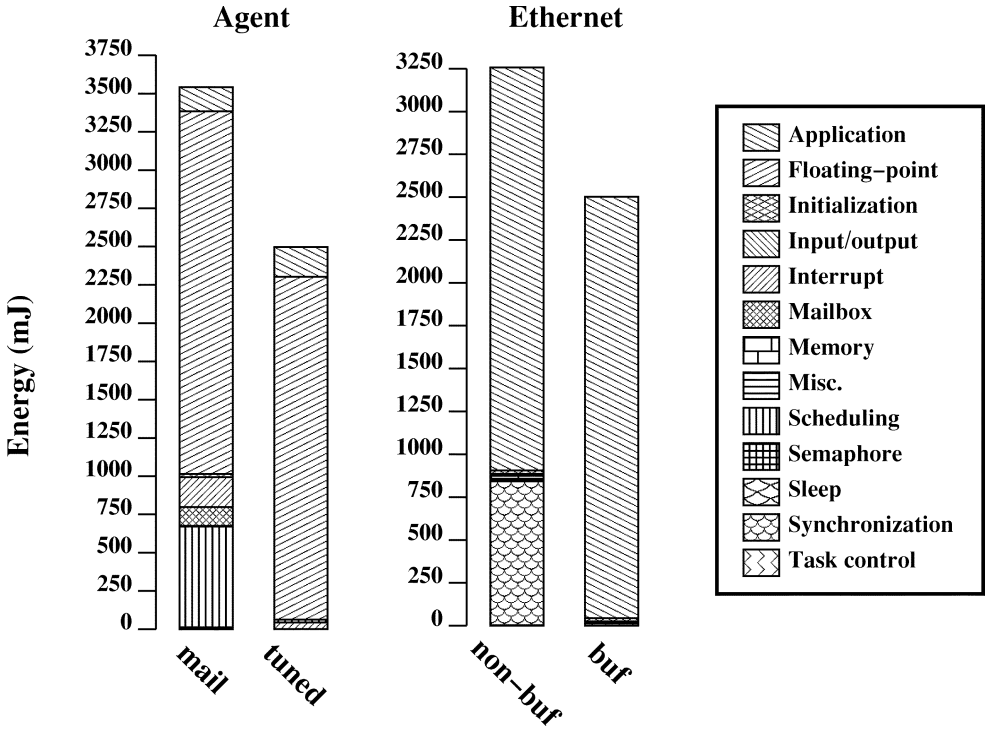
In our third example, we consider checksum computation and interfacing with an Ethernet controller that has high per-access overhead. This action occurs at the lowest level of a TCP/IP protocol stack. Incoming packets are processed to derive their checksums. The packets are subsequently transmitted to the output device.

The most straightforward implementation of this algorithm, shown in Fig. 6(a), processes each packet as soon as it is available. However, in this example, preparing the Ethernet controller to receive a packet, represented by the Procure Ethernet controller operation in Fig. 6(a), is costly. The nonbuf bar in Fig. 5(b) shows the energy consumed by this straightforward implementation, broken down by RTOS service and application categories.

It is possible to amortize the cost of Procure Ethernet controller over the transmission of multiple packets by decoupling packet generation from transmission to the Ethernet controller. In this energy-optimized implementation, the application is broken into three tasks, as shown in Fig. 6(b). The Checksum computation task communicates packets to the Buffer management task via shared memory. When the Buffer management task has enqueued a number of packets, it transfers them simultaneously to the Output task that procures the Ethernet controller and transmits all the packets in its queue.

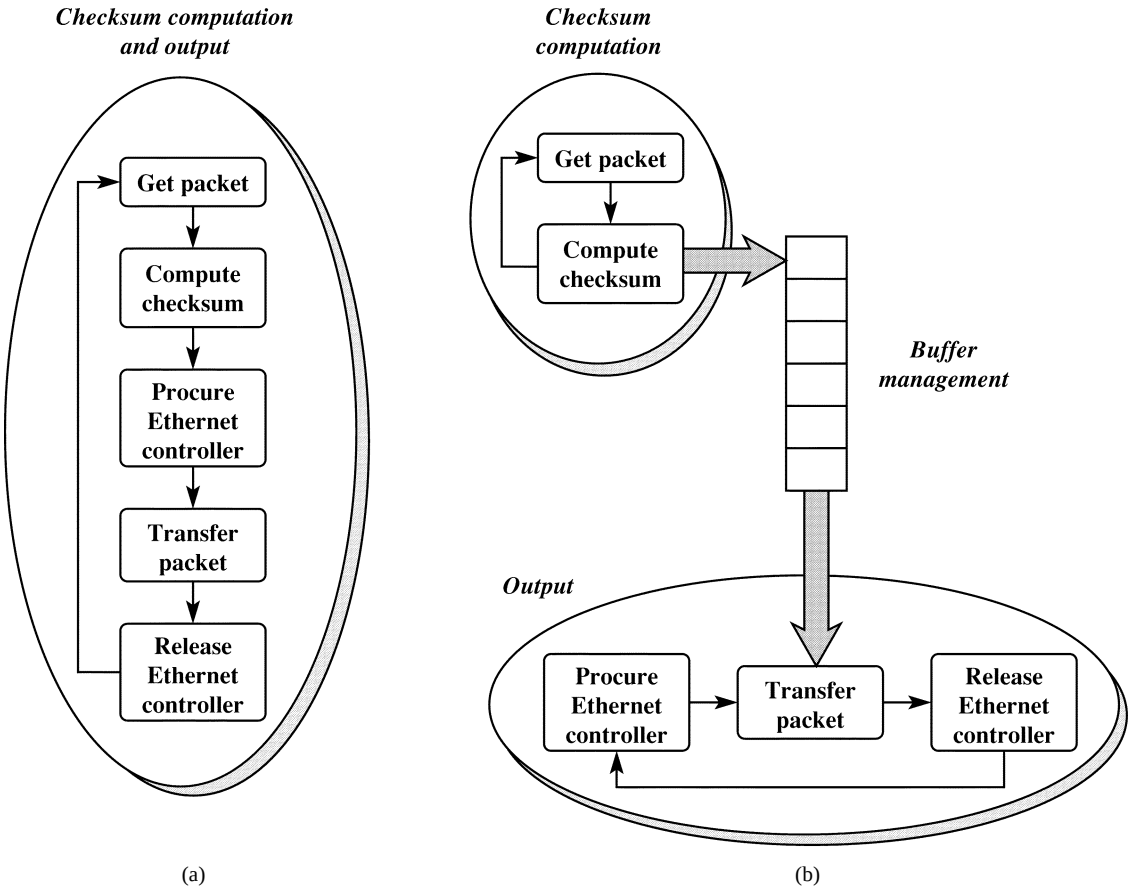
The buf energy bar in Fig. 5(b) shows the energy consumed by the energy-optimized version of the Ethernet interface example. Although some energy or time is consumed by functions in each of the classifications listed in the key, some of these classifications account for very little energy or time consumption, and are barely visible in the bar charts.

Energy optimization of the Ethernet interface example results in a 23.1% overall decrease in energy consumption, with most of the savings resulting from reduced reliance on hardware access synchronization and initialization services. Power consumption reduced by 0.1%, i.e., the energy savings resulted from a reduction in execution time, not average power consumption. The energy saved in the hardware access synchronization and initialization services was sufficient to more than offset a 2.9% increase in energy resulting from the increased com-



(a) (b)

Fig. 5. (a) Commodity trading agent example energy and (b) Ethernet interface example energy by RTOS service category.



(a) (b)

Fig. 6. (a) Straightforward implementation and (b) multiprocess implementation of the Ethernet interface example.

plexity of the multiple-task implementation. One could easily convert some of these energy savings into power savings by putting the processor and memory into sleep mode for the amount of time saved in the buffered version. In this example, the RTOS consumed only 1.2% of the overall energy in the version that was not energy-optimized, and a similar percentage of overall energy in the energy-optimized version.

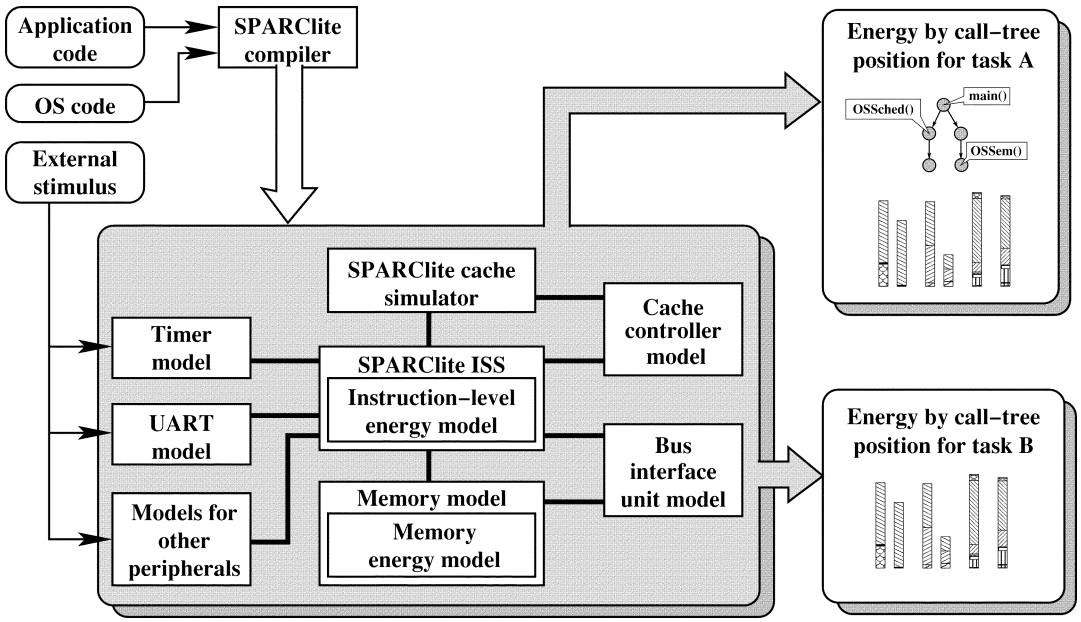


Fig. 7. Energy analysis framework.

However, in a number of other examples shown in Section V, the RTOS consumes a larger portion of the embedded system's energy.

The examples presented in this section demonstrate that the manner in which an RTOS power analysis infrastructure may be used to determine promising areas for power optimization and evaluate the tradeoffs between power and other costs. Understanding the effects of RTOS use on time and energy allows a designer to better optimize the these energy embedded system attributes.

IV. Energy Analysis Infrastructure

In this section, we present our RTOS energy analysis framework. We first describe the inputs and outputs of our framework. Next, we present a high-level view of its building blocks, and the manner in which they interact to analyze the system energy consumption. We then present some details of individual building blocks.

A. Inputs and Outputs

Our framework can be used to analyze the energy consumption of an application, consisting of multiple tasks, executing under a multitasking operating system. These tasks interact with each other, as well as with peripheral devices such as universal asynchronous receivers and transmitters (UARTs), brake sensors, and other hardware components. The embedded system is simulated to obtain a detailed report of the energy consumed by different application/RTOS functions.

Fig. 7 depicts our energy analysis framework. The application, which consists of multiple processes, is compiled and linked together with the μC/OS-II RTOS and Fujitsu's SPARClite runtime libraries. In addition, a model of the system's environment or external stimuli is provided to our framework.

The outputs of our software, shown at the right of Fig. 7, include call-trees for each task and the RTOS. Each call-tree node corresponds to a function call, and has a child node for each function call instance that occurs within it. An edge from function foo to function bar indicates that foo calls bar. The nodes of the call-tree are annotated with the functions they represent, and the energy and time consumed by each invocation of the function. The contributing sources of energy consumption within the function, e.g., instruction execution, stalls, DRAM refreshing, are recorded. Note that if a function h is called from two functions f and g, we create separate nodes in the call-tree corresponding to these two scenarios. This ensures that the energy consumption statistics of a function are separated by caller. Each call instance's energy information can be examined separately or the call-instances may be combined in order to find the total energy consumed by all of the instances of a function located at a given position in the call-tree. At each position in the call-tree, detailed information is reported about the sources of energy consumption within the function. In addition, a total hierarchical energy consumption, equal to the sum of the total energy consumptions of a node's children, is given.

Table I shows a portion of the automatically formatted output of the system when analyzing a semaphore example. In this example, concurrent tasks are synchronized through the use of RTOS services. We present this table in order to give the reader a concrete idea of the sort of output the embedded system power analysis tool produces. Note that each context, e.g., realstart and Task1, is a separate start node in the call-tree hierarchy. The same function may appear more than once in the call-tree, if it is called from different locations, e.g., the window underflow trap service routine win_unf_trap in Task1. Although only energy per invocation, percentage of total energy, total time, and number of calls are displayed in this table, the analyzer also produces more detailed reports on embedded system attributes, e.g., it can separate energy consumption into sleep energy, stall energy, cache stall energy, memory access energy, memory idle energy, and instruction processing energy.

For the sake of brevity, the call-tree has been pruned to limit its depth and breadth. It truncates the call-tree at a depth of three and omits the Task2 context. For example, the table shows information about the realstart and Task1 contexts. Task1 calls OSSemPend that in turn calls a number of other functions, including OSSched. Although OSSched calls other functions, they are omitted from the table for brevity. OSSemPend consumed 104.59 mJ, including the energy consumed by all of the other functions it calls. OSSched consumed 66.44 mJ per invocation and it is invoked 999 times at this position in the call-tree. Including the energy of the other functions it calls, it consumes 6.35% of the total system energy and executes for a total of 51.95 ms. Note that the figure produced by multiplying the energy consumption of each one of the functions OSSemPend calls by the number of times the function is called is slightly lower than OSSemPend's total energy consumption. The difference between these figures is the amount of energy consumed by instructions at OSSemPend's position in the call-tree.

TABLE I
HIERARCHICAL CALL-TREE FOR THE SEMAPHORE EXAMPLE

		Function	Energy(μJ) invocation	Energy (%)	Time (ms)	Calls
realstart 25.40 mJ total 2.43 %	init_tvecs		1.31	0.00	0.00	1
	init_timer 18.01 mJ total 1.72 %	liteled	4.26	0.00	0.00	1
	startup 7.39 mJ total 0.71 %	do_main	7363.11	0.70	5.57	1
		save_data	5.08	0.00	0.00	1
		init_data	4.23	0.00	0.00	1
		init_bss	2.86	0.00	0.00	1
		cache_on	8.82	0.00	0.01	1
Task1 508.88 mJ total 48.69 %	win_unf_trap		6.09	1.16	9.43	1999
	OSDisableInt		0.98	0.09	0.82	1000
	OSEnableInt		1.07	0.10	0.92	1000
	OSSemPend 104.59 mJ total 10.01 %	win_unf_trap	6.00	0.57	4.56	999
		OSDisableInt	0.94	0.18	1.56	1999
		OSEnableInt	0.94	0.18	1.56	1999
		OSEventTaskWait	13.07	1.25	9.89	999
		OSSched	66.44	6.35	51.95	999
	OSSemPost 9.82 mJ total 0.94 %	OSDisableInt	0.96	0.09	0.78	1000
		OSEnableInt	0.98	0.09	0.81	1000
	OSTimeGet 4.62 mJ total 0.44 %	OSDisableInt	0.84	0.08	0.66	1000
		OSEnableInt	0.98	0.09	0.81	1000
	CPUInit 0.29 mJ total 0.03 %	BSPInit	3.52	0.00	0.00	1
		exceptionHandler	15.51	0.02	0.17	15
	printf 368.07 mJ total 35.22 %	win_unf_trap	6.18	0.59	4.87	1000
		vfprintf	355.04	33.97	257.55	1000

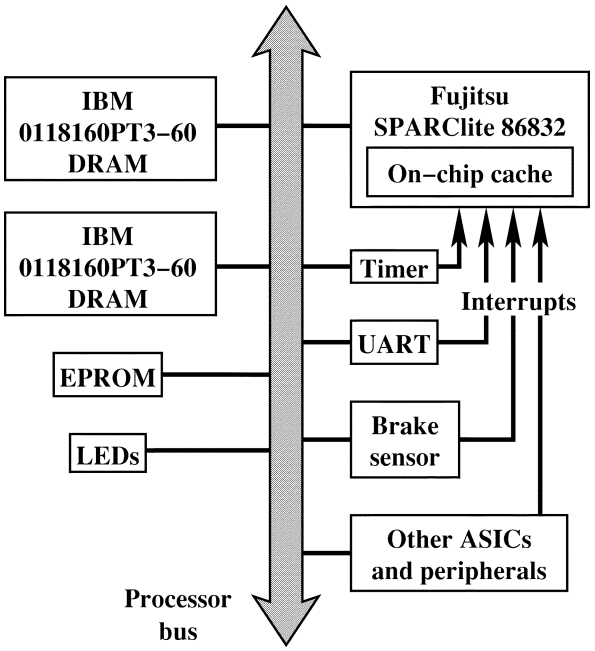


Fig. 8. Modeled architecture.

B. System Overview

We now describe the operation of our energy analysis framework. The simulated embedded system consists of a processor interacting with a set of application-specific integrated circuits (ASICs) and other peripherals. As shown in Fig. 8, our energy analysis infrastructure models a Fujitsu SPARClite processor, connected to two fast page-mode DRAMs, a timer, a UART, and a number of other peripherals. Cycle-accurate simulators have a reputation for being slow. However, this approach is sufficiently fast to handle substantial applications; a similar simulation infrastructure subsequently built by colleagues booted Linux in less than 5 min on a Pentium III processor running at 667 MHz [34].

In order to analyze the energy consumption of the system, we need functional models and energy models of its constituent parts. Instruction-level power models for the Fujitsu SPARClite processor and internal cache can be found in the literature [13]. The internal operation of the SPARClite processor is simulated using an instruction set simulator (ISS) [18] that we have modified so that it handles interaction with other components in the modeled embedded system. We have implemented an easy-to-use, object-oriented, inheritance-based method of adding new hardware to the simulated system, e.g., the brake sensors used in the ABS example. Application-specific devices may interrupt the operation of the processor. We use interrupt routines based on those found in the Fujitsu MB86832 evaluation kit, and $\mu C/$OS-II. Applications run under $\mu C/$OS-II. The addition of hardware interrupts to the embedded system simulator required significant changes to maintain correct simulation. In particular, it is not possible to use offline hardware models in the presence of coprocessor generated interrupts.

The ISS simulates the cycle-by-cycle execution of the processor, i.e., it accounts for effects such as branch delays, pipeline flushes, control-flow mispredictions, etc. We have enhanced this ISS in a number of ways. In order to account for the effects of cache misses, we added an online cache simulator designed specifically to model the SPARClite processor's cache. It is necessary to use an online cache simulator in order to know, during execution, whether or not a cache miss has occurred. An offline cache simulator would not allow the correct simulation of an embedded system because, due to races with interrupts generated by other peripherals, the presence or absence of a miss penalty may change the flow of execution. The cache simulator accounts for the cache and memory behavior. We model a number of SPARClite-specific features. Among these, low-power sleep mode is particularly important. In addition, we model external memory. Specifically, we simulate the cache and on-board bus interface unit of a Fujitsu MB86832 [37], [38], as well as the operation of two IBM0118160PT3–60 low-power fast page-mode DRAMs [39]. Memory-energy consumption is derived from the manufacturer's data-sheet, and depends on the DRAM's mode of operation. We consider the energy required to drive the processor-memory bus. Our power model is built from datasheets [39] and published current

measurements [13]. If the hardware implementation of an additional device a designer wants to integrate into the system is known, its energy consumption can be computed using known energy analysis techniques [8], [9], [12].

As mentioned earlier, our energy analysis framework organizes energy-consumption data by function. Therefore, in addition to evaluating the energy consumed by the system in a cycle, our energy analyzer needs to keep track of the function and process that are currently being executed. In general, the manner in which the context is determined is specific to the operating system, and the processor being considered. μC/OS-II performs scheduling and context switch occurs through the function OSSched. Our framework uses this information to keep track of context switches. Function calls are performed using the jmpl instruction from the SPARC assembly language. The name of the function to which control flow is transferred is determined from the symbol table that associates an address with each function and global variable. The problem of tracking returns from function calls is complex and requires information specific to the instruction set architecture of the processor being used, the manner in which the compiler translates different control-flow constructs in the high-level programming language into assembly code, and information specific to the RTOS code that performs context switching.

Our energy analysis technique is nonintrusive. This differs with many well-known software debugging and performance analysis techniques that augment the program to be analyzed with monitoring code in order to enhance observability of the program state and internals. While the addition of monitoring code eases analysis, it results in a loss of accuracy because the monitoring code modifies the parameters that needs to be measured: execution time and energy. Additionally, this extra code may change the order in which tasks execute in an embedded system containing multiple hardware devices. The need to perform cycle-accurate performance analysis is heightened in the presence of external devices that communicate with the processor. Inaccuracies in timing can cause a change in the functionality of the system being implemented, leading to inaccurate control-flow and energy results. Since we use cycle-accurate processor and cache energy models, our framework does not suffer from this problem. When run on a 336-MHz UltraSPARC-II with 4 Gb of memory, the simulator takes approximately 40 min to simulate the 14-s original version (i.e., nongate) of the ABS example and approximately 12 min to simulate the 2.5-s original version (i.e., nonbuf) of the Ethernet interface example.

There is one caveat regarding the power model used for the SPARClite processor. We selected the Fujitsu SPARClite MB86832 for simulation because an evaluation kit for this processor is currently available from Fujitsu, allowing us to use their development tool's electrically programmable read-only memory (EPROM) code to facilitate the simulation of a concrete embedded system. However, we do not currently have a power model for the MB86832. We used the instruction-level power model for the Fujitsu SPARClite MB86934 [13]. The core clock frequency for the modeled processor is 80 MHz, while the core clock frequency used to build the power model is 20 MHz. The input–output (I/O) clock frequency for the modeled processor is 26.7 MHz, while the core clock frequency used to build the power model is 10 MHz. It was necessary to scale the current values in the power model in order to account for the increased clock frequencies. According to the MB86832 data-sheet, this processor's current scales linearly with clock frequency [38]. This behavior is to be expected for conventional, low-leakage CMOS processes. The instruction-level power model for the MB86934 does not separate the power consumed in the processor core from the power consumed in the I/O circuits. Therefore, we used the 20-MHz core and 10-MHz I/O MB86934 datasheet to determine the relative power contributions of the processor core and I/O circuits. We then, under the assumption of a linear relationship between clock frequency and power consumption, determined the contribution of processor core and I/O power consumption for the 80-MHz core and 26.7-MHz I/O MB86832.

C. System Details

In this section, we describe the operation of two key components of our target system architecture: the processor and the operating system. We first present an overview of the processor, and then briefly describe the μC/OS-II RTOS.

Our system is built around a Fujitsu SPARClite MB86832, a 32-bit RISC processor, operating at 80 MHz, with an external bus speed of 26.7 MHz. It implements a superset of the SPARC v8 instruction set architecture. Its integer unit has a five-stage pipeline that can handle data interlocks and a branch handler to perform control-flow transfers efficiently. The bus interface unit is capable of providing single-cycle access to the on-chip cache. The processor has 136 registers, organized into eight overlapping register windows, and 8-KB instruction and data caches. Multiply and divide operations are supported by dedicated, on-chip hardware that can complete 32-bit multiplications in five cycles. The processor also has a power-down mode that can be employed to reduce energy consumption.

We have taken care to simulate the context-dependent IBM0118160PT3–60 memory and MB86832 bus interface unit timing in sufficient detail to ensure that memory accesses require the number of cycles implied by the timing diagrams in the specifications. In addition, we simulate stalls resulting from periodic distributed DRAM refreshes.

μC/OS-II is Jean Labrosse's portable real-time kernel for microprocessors and micro-controllers. We use the version Brad Denniston ported to the MB86832 processor. μC/OS-II has been used in many commercial applications, and its performance is comparable to that of other commercial RTOSs. μC/OS-II supports multitasking, and can handle up to 63 concurrent processes. The kernel is fully preemptive. The RTOS is designed to be scalable, i.e., designers who do not require some of its features may save memory by easily building a lightweight version of μC/OS-II. The RTOS provides a number of services such as scheduling, task management, interprocess communication, memory management, interrupt handling, and timer-related services. We chose μC/OS-II for our experiments because it is modular, well-designed, and well-documented; its source code is readily available. Further information on μC/OS-II can be found on the Internet at http://www.uCOS-II.com, or in Labrosse's book [3].

D. Extending Our Approach to Other Embedded Systems

Our approach for analyzing RTOS and application software power consumption can be extended to other processors and operating systems. However, there are system-dependent components in this approach.

It is necessary to have ISSs for the processors used in the target embedded system. There must be a method for tracing the status of the simulated processor cycle by cycle, in order to record energy consumption, detect context switches, and simulate interaction with other hardware in the embedded system. Although it is conceivable for an ISS to provide a runtime interface meeting these requirements, it is our belief that, in practice, the ISS source code will be required. ISSs are available for a number of popular architectures. Vendors sometimes provide simulators for more exotic processors. A designer who wants to use our power analysis method on complex processors for which ISSs are not available will face a substantial burden. Fortunately, getting access to simulation modules for system-specific ASICs is likely to be straightforward, as the in-house simulators used to design and debug the ASICs are likely to be available.

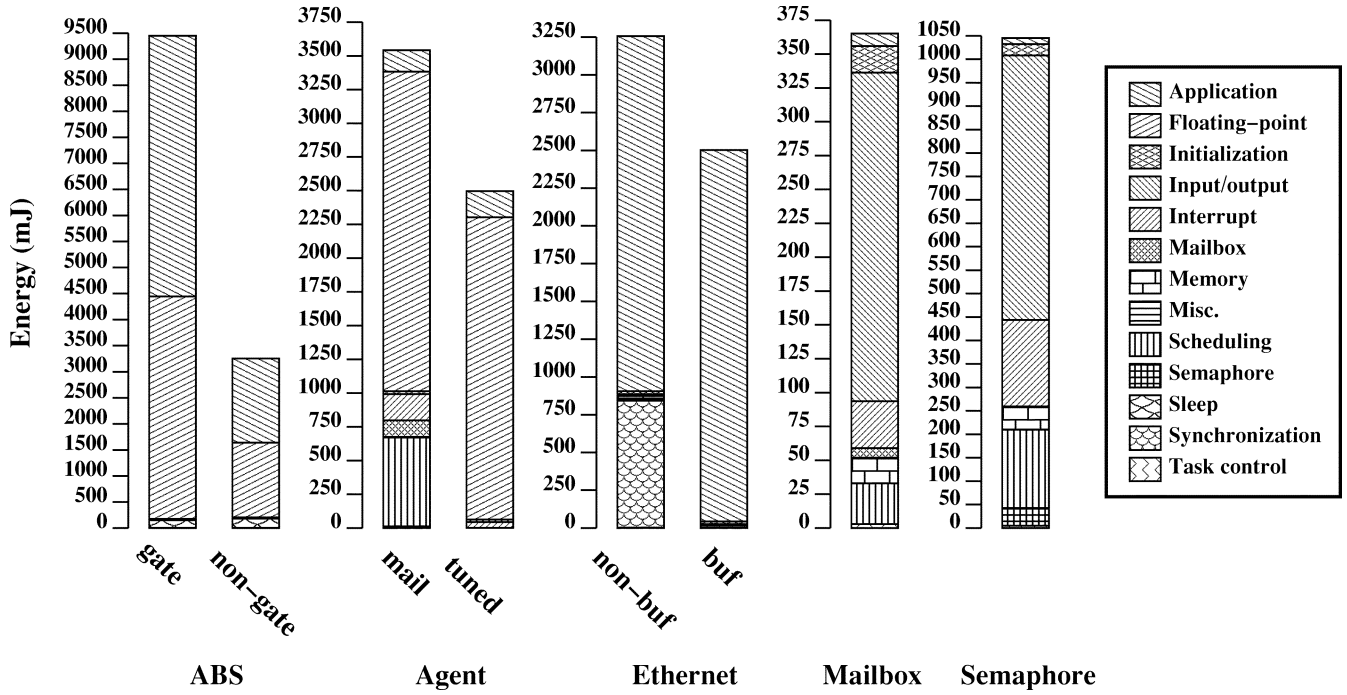


Fig. 9. Energy consumption profiles.

Unless power consumption was a primary consideration in RTOS design, minor changes to an RTOS can significantly improve its power consumption characteristics. A feature of μC/OS-II provides support for this observation. When no user-defined processes are running, an idle task executes. Normally, this task repeatedly increments a variable. By comparing the actual number of increments in a given time-span with the maximum number of increments possible in that time-span, μC/OS-II keeps track of the percentage of time spent idle. This behavior is beneficial, as long as one is not trying to minimize power consumption. There are sophisticated approaches one could use to dramatically reduce idle-power consumption. However, even the straightforward expedient of preventing the variable from being incremented eliminates numerous writes to the processor's write-through cache, thereby reducing memory power consumption. The ability to make changes to the source code of an RTOS increases the designer's flexibility in optimizing embedded system power consumption. However, even if the source code is not available, our approach allows a designer to modify an application's use of RTOS services in order to reduce power consumption. Note that, even if an RTOS's source code is not available, as long as its method of switching contexts can be determined, the approach presented in this paper can be used.

Finally, it is necessary to have power models for the embedded system devices that consume a significant amount of power. It is our hope that, in the future, hardware vendors will see the competitive advantage of providing customers with detailed power information about their products. Until this practice becomes common, designers who want to apply our approach will be forced to rely on power models and analysis techniques found in the literature [8], [9], [12], [13], internally developed power models, or the limited power information found in conventional data-books. Note that, for some processors, this power information is sufficient to allow a reasonable estimate of power consumption.

V. Results and Case Studies

We analyzed the energy consumption of μC/OS-II RTOS when running several embedded applications. In all cases, we targeted the Fujitsu SPARClite processor-based embedded system presented in Section IV-B. Some examples are portions of real embedded system application software, some were designed to illustrate design alternatives, and others were included with the RTOS distribution as illustrative example. Overall, care was taken to ensure that key RTOS functions and services were used by the chosen applications.

For each example, we categorized energy consumption by RTOS, board support package, and application service type, as explained in the following list.

- **Application**: Non-RTOS functions;
- **Floating-point**: Integer operations to simulate floating point mathematics;
- **Initialization**: Embedded system initialization functions. This is typically executed only once during an application's run;
- **Input/output**: Input and output formatting and communication with the system's UART channels;
- **Interrupt**: Interrupt service routines;
- **Mailbox**: Code to handle task communication with mailboxes;
- **Memory**: Memory initialization, allocation, and copying functions;
- **Misc.**: Functions not in other categories;
- **Scheduling**: Task scheduling;
- **Semaphore**: Semaphore-based task synchronization code;
- **Sleep**: Sleep mode;
- **Synchronization**: Nonsemaphore-based task synchronization code;
- **Task control**: Task management, e.g., task creation.

Fig. 9 shows the energy consumed by different RTOS, board support, and application services. Each vertical bar represents a distinct example. Vertical bars are divided to indicate functions. For instance, in the mailbox example, I/O primitives used by the RTOS account for a larger portion of the energy consumption than any other function category. Fig. 10 presents a similarly formatted characterization of time consumption by RTOS service and function category.

The ABS, Agent, and Ethernet examples are described in Section III. The ratio of processor energy consumption to DRAM energy consumption varied from 2.71 (the energy-optimized version of the Ethernet

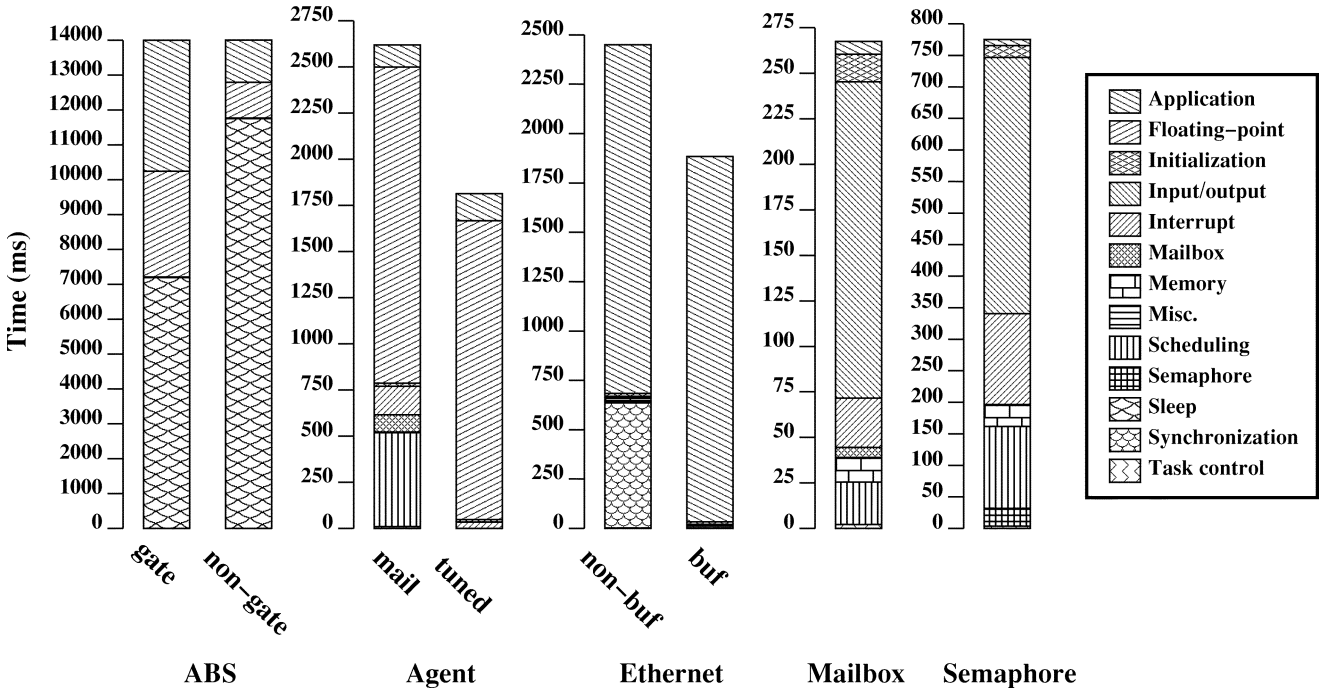


Fig. 10. Time consumption profiles.

interface example) to 2.94 (the energy-optimized version of the ABS example). The results in this section, and in Section III, indicate that an embedded system's RTOS and board support routines may be directly responsible for a significant portion of the embedded system's energy consumption. The percentage of system energy directly consumed by the RTOS and board support routines may vary dramatically from approximately 1% (the energy-optimized version of the Ethernet interface example) to 99% (the mailbox example), depending on how heavily the software relies on RTOS services. Even when the RTOS does not directly consume a significant percentage of the system's energy, one can significantly reduce overall energy consumption by more wisely using RTOS services, as demonstrated by the different versions of the ABS example.

The mailbox example illustrates the use of mailboxes for interprocess communication. It consists of three application tasks that communicate via the shared memory mailbox communication service provided by μC/OS-II. The tasks also perform writes to the UART. Fig. 9 shows that, in this example, the main sources of energy consumption are input/output primitives, interrupt service routines, task scheduling, as well as RTOS and processor initialization code. Mailbox management services also consume a small but significant fraction of the system's energy. Formatting and transmitting data to the UART can be energy-intensive, and should be sparingly used in an energy-constrained implementation. The application code relies heavily on RTOS and board support routines. As a result, the application code only consumes 1.0% of the total system energy, with RTOS and board support services consuming the other 99.0%.

In the semaphore example, concurrent tasks are synchronized through the use of RTOS services. RTOS primitives that post and release semaphores account for a small but significant portion of the system's energy consumption. The application code consumed 1.2% of the total system energy, with RTOS and processor support services consuming the other 98.8%.

From the results presented above, one can observe that the embedded system consumed significantly less power during sleep mode (14.2–18.0 mW depending on example) than when running in other modes. As described in Section IV, a call-tree node holds the total time and energy of all function calls located at a given position in the call-tree. The average power consumption of call-tree nodes, i.e., context-dependent function execution, varied from 769 mW (OSEnableInt) to 1,047 mW (uart_delay). However, the differences among the power consumption of RTOS service classes were smaller. Average RTOS service class power consumption varied from 842 (for interrupt service routines) to 976 mW (for floating-point routines). While there was a strong correlation between execution time and energy consumption for the examples in which sleep mode was not used, it would be unwise to generalize this observation to all embedded systems. In embedded systems containing peripheral processors that consume a substantial amount of energy, and whose control is relegated to a subset of the RTOS service classes, there would be substantial differences between the power consumptions of different RTOS service and function categories.

Table II shows the minimum and maximum energy per invocation for each RTOS service, board support package routine, and standard library routine used in our examples. These routines might consume less energy than the minimum in the table, or more energy than the maximum in the table, if they are used in a manner not encountered in any of our examples. However, for applications similar to our examples, these values provide a reasonable range for the energy costs of RTOS services and other support routines.

VI. Conclusion and Recommendations

In this paper, we have described the design and implementation of an RTOS power analysis infrastructure. Examples were presented to illustrate the use of this infrastructure. By analyzing a commercial RTOS, μC/OS-II, running several applications, we have demonstrated that the manner in which the RTOS is used has a significant impact on an embedded system's power consumption. Insights derived from such RTOS power analysis may be used to optimize embedded software power consumption and drive research on high-level power modeling of different RTOS components. Furthermore, this work enables power-efficient RTOS and application design, and may be incorporated into power-aware system-level design tools.

TABLE II
RTOS SERVICE ENERGY PER INVOCATION

Service	Minimum energy (μJ)	Maximum energy (μJ)	Service	Minimum energy (μJ)	Maximum energy (μJ)
AgentTask	3.41	4727.88	fptodp	17.46	49.72
BSPInit	3.52	3.52	fstat	16.34	16.34
CPUInit	287.15	287.15	fstat_r	31.26	31.26
GetPsr	0.38	0.55	init_bss	2.86	3.07
GetTbr	0.40	0.53	init_data	4.23	4.37
InitTimer	2.53	2.53	init_timer	18012.10	20347.00
OSCtxSw	46.63	65.65	init_tvecs	1.31	1.31
OSDisableInt	0.84	1.31	isatty	1.82	1.82
OSEnableInt	0.84	1.31	liteled	4.26	4.26
OSEventTaskRdy	26.45	29.16	litodp	10.22	225.41
OSEventTaskWait	11.62	13.20	localeconv	2.08	13.82
OSEventWaitListInit	30.35	31.06	localeconv_r	0.56	0.92
OSInit	7057.43	7057.43	lshrdi3	2.63	3.37
OSMboxCreate	41.12	43.25	make_dp	9.87	40.44
OSMboxPend	10.11	130.59	malloc_r	73.66	73.66
OSMboxPost	7.78	129.06	mbtowc	3.21	4.09
OSMemInit	4432.06	4432.06	memchr	2.15	16.38
OSQInit	60.02	60.02	memmove	3.41	18.45
OSSched	10.24	80.73	morecore_r	57.20	57.20
OSStartHighRdy	20.53	20.53	pack_d	6.01	24.65
OSTCBInit	42.55	45.15	pack_f	3.49	7.66
OSTaskCreate	84.29	87.98	printf	849.19	1054.54
OSTaskCreateExt	2145.03	2145.03	putCharPort1	19.43	32.87
OSTaskCreateHook	1.94	1.94	rand	2.47	3.15
OSTaskStkInit	16.56	31.76	rand_range	912.52	996.73
OSTaskSwHook	0.58	1.13	rdtbr	0.38	0.53
Roulette	926.92	5684.69	rint	3.76	435.11
agent_broadcast	957.72	4714.15	save_data	5.08	5.08
agent_buy	7.24	8.94	sbrk	5.00	19.06
agent_init	71.19	211.09	sbrk_r	7.22	33.27
agent_offer	241.37	1279.00	sfvwrite	50.31	530.08
agent_price	228.41	830.43	sinit	35.90	35.90
agent_sell	6.26	933.14	sitofp	7.67	86.79
cache_off	3.18	3.18	smakebuf	131.77	131.77
cache_on	8.68	8.68	sprint	53.60	533.44
do_global_ctors	3.26	3.26	std	9.09	9.09
dpadd	31.31	139.92	swrite	467.56	498.66
dpdiv	237.57	291.14	swsetup	138.95	138.95
dpsub	41.59	286.74	uart_delay	14.39	14.80
dptoli	8.44	17.03	unpack_d	5.24	8.59
exceptionHandler	15.26	18.80	unpack_f	3.60	6.10
fflush	477.04	507.35	vfprintf	837.02	1036.37
fpadd_parts	3.79	255.83	vfprintf_r	829.07	1022.72
fpdiv	21.03	72.81	win_ovf_trap	11.25	12.11
fpdiv_parts	4.23	261.22	win_unf_trap	6.00	11.84
fpmul	22.00	40.66	write	461.03	468.38
fpmul_parts	4.73	18.07	write_r	463.92	482.99

Based upon our observations, we have found a few general guidelines that designers should follow in order to use an RTOS in a power-efficient way. However, before presenting these guidelines, we must first make a few caveats. The most power-efficient implementation of embedded system software is processor-dependent and RTOS-dependent. We strongly suggest implementing or simulating a prototype before expending heroic efforts on low-level power optimization. One should start trading off code flexibility and maintainability for power efficiency only after it is clear, e.g., via the type of energy profiling described in this paper, which portion of the RTOS, board support package, or application code is unnecessarily consuming power. The guidelines we present, here, are no substitute for using a detailed power analysis infrastructure, of the sort presented in this paper, during the design of an embedded system.

A number of energy reduction options are available to an embedded system designer with access to an RTOS, as follows.

- Rewrite high energy consumption portions of an application to avoid unnecessary use of the RTOS scheduler.
- When synchronization between tasks is implicitly carried out, do not use RTOS services to do (redundant) synchronization. This may be easier said than done because redundant synchronization can make code more robust.
- Take advantage of RTOS primitives, e.g., process support, to allow easy implementation of multiprocess schemes that amortize the costs of high-overhead operations.
- If power analysis indicates that memory management consumes a substantial proportion of embedded system power, consider custom, e.g., uniform block, memory management for commonly allocated and deallocated data types.
- Concentrate on special modes available in the processor. Most designers already pay some attention to code execution time and, in the absence of special processor modes, there is a strong correlation between execution time and energy for general-purpose processors. However, using special processor modes, e.g., sleep mode, can dramatically reduce power consumption. One can leverage an RTOS to easily retrofit an existing application for power reduction, e.g., one may use a low-priority task that puts a processor into sleep mode.

We emphasize that the above recommendations are not exhaustive; they will not be beneficial for every embedded system. Our strongest suggestion is to examine an embedded system's RTOS/application energy profile before attempting to power-optimize code.

ACKNOWLEDGMENT

The authors would like to thank Dr. L. French, from NEC C&C Research Labs, for helpful discussions on real-time operating systems and his assistance with the Ethernet interface example.

REFERENCES

[1] S. Heath, *Embedded Systems Design*. Boston, MA: Butterworth–Heinemann, 1997.
[2] J. J. Labrosse, *Embedded Systems Building Blocks*. Lawrence, KS: R & D Books, 1997.

[3] ——, *MicroC/OS-II*. Lawrence, KS: R & D Books, 1998.
[4] P. A. Laplante, *Real-Time Systems Design and Analysis: An Engineers Handbook*. Piscataway, NJ: IEEE Press, 1993.
[5] R. Sharma, "Distributed application development with inferno," in *Proc. Design Automation Conf.*, June 1999, pp. 146–150.
[6] D. Stepner, N. Rajan, and D. Hui, "Embedded application design using a real-time OS," in *Proc. Design Automation Conf.*, June 1999, pp. 151–156.
[7] W. Warner, "Non-pre-emptive multithreading performs embedded software's juggling act," *Electron. Design News*, vol. 44, pp. 117–126, July 8, 1999.
[8] L. Benini and G. De Micheli, *Dynamic Power Management: Design Techniques and CAD Tools*. Norwell, MA: Kluwer, 1997.
[9] A. R. Chandrakasan and R. W. Brodersen, *Low Power Digital CMOS Design*. Norwell, MA: Kluwer, 1995.
[10] G. Yeap, *Practical Low Power Digital VLSI Design*. Norwell, MA: Kluwer, 1998.
[11] J. Monteiro and S. Devadas, *Computer-Aided Design Techniques for Low Power Sequential Logic Circuits*. Norwell, MA: Kluwer, 1996.
[12] J. Rabaey and M. P., Eds., *Low Power Design Methodologies*. Norwell, MA: Kluwer, 1996.
[13] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: A first step toward software power minimization," *IEEE Trans. VLSI Syst.*, vol. 2, pp. 437–445, Dec. 1994.
[14] T. Sato, Y. Ootaguro, M. Nagamatsu, and H. Tago, "Evaluation of architecture-level power estimation for CMOS RISC processors," in *Proc. Symp. Low Power Electron.*, Oct. 1995, pp. 44–45.
[15] C. T. Hsieh, M. Pedram, G. Mehta, and F. Rastgar, "Profile-driven program synthesis for evaluation of system power dissipation," in *Proc. Design Automation Conf.*, June 1997, pp. 576–581.
[16] L. Benini and G. De Micheli, "System-level power optimization: Techniques and tools," in *Proc. Int. Symp. Low Power Electron. Design*, Aug. 1999, pp. 288–293.
[17] B. Dave, G. Lakshminarayana, and N. K. Jha, "COSYN: Hardware-software co-synthesis of embedded systems," in *Proc. Design Automation Conf.*, June 1997, pp. 703–708.
[18] 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. 188–193.
[19] 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.
[20] M. Lajolo, A. Raghunathan, S. Dey, L. Lavagno, and A. Sangiovanni-Vincentelli, "Efficient power estimation techniques for HW/SW systems," in *Proc. Alessandro Volta Memorial Wkshp. on Low Power Design*, Mar. 1999, pp. 191–199.
[21] S. Gurumurthi, A. Sivasubramaniam, M. J. Irwin, N. Vijaykrishnan, M. Kandemir, T. Li, and L. K. John, "Using complete machine simulation for software power estimation: The softwatt approach," in *Proc. Int. Symp. High-Performance Comput. Architecture*, Feb. 2002, pp. 141–150.
[22] V. Tiwari, S. Malik, and A. Wolfe, "Compilation techniques for low energy: An overview," in *Proc. Symp. Low Power Electronics*, Oct. 1994, pp. 38–39.
[23] T. Simunic, G. De Micheli, and L. Benini, "Energy-efficient design of battery-powered embedded systems," in *Proc. Int. Symp. Low Power Electron. Design*, Aug. 1999, pp. 212–217.
[24] J. L. da Silva, F. Catthoor, D. Verkest, and H. De Man, "Power exploration for dynamic data types through virtual memory management refinement," in *Proc. Int. Symp. Low Power Electron. Design*, Aug. 1998, pp. 311–316.
[25] Q. Qiu, Q. Wu, and M. Pedram, "Stochastic modeling of a power-managed system: Construction and optimization," in *Proc. Int. Symp. Low Power Electron. Design*, Aug. 1999, pp. 194–199.
[26] L. Benini, A. Bogliolo, S. Cavallucci, and B. Ricco, "Monitoring system activity for OS-directed dynamic power management," in *Proc. Int. Symp. Low Power Electron. Design*, Aug. 1998, pp. 185–190.
[27] I. Hong, D. Kirovski, G. Qu, M. Potkonjak, and M. B. Srivastava, "Power optimization of variable voltage core-based systems," in *Proc. Design Automation Conf.*, June 1998, pp. 176–181.
[28] T. Ishihara and H. Yasuura, "Voltage scheduling problem for dynamically variable voltage processors," in *Proc. Int. Symp. Low Power Electron. Design*, Aug. 1998, pp. 197–202.
[29] T. Pering, T. Burd, and R. Brodersen, "The simulation and evaluation of dynamic voltage scaling algorithms," in *Proc. Int. Symp. Low Power Electron. Design*, Aug. 1998, pp. 76–81.
[30] N. K. Jha, "Low power system scheduling and synthesis," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 2001, pp. 259–263.
[31] *Proc. Workshop Compilers Oper. Syst. Low Power*, L. Benini, M. Kemir, and J. Ramanujam, Eds., Sept. 2002.
[32] A. Vahdat, A. R. Lebeck, and C. S. Ellis, "Every Joule is precious: A case for revisiting operating system design for energy efficiency," in *Proc. ACM SIGOPS Eur. Workshop*, Sept. 2000, pp. 31–36.
[33] 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.
[34] T. K. Tan, A. Raghunathan, and N. K. Jha, "EMSIM: An energy simulation framework for an embedded operating system," in *Proc. Int. Symp. Circuits Syst.*, May 2002, pp. 464–467.
[35] K. Baynes, C. Collins, E. Fiterman, B. Ganesh, P. Kohout, C. Smit, T. Zhang, and B. Jacob, "The performance and energy consumption of three embedded real-time operating systems," in *Proc. Int. Conf. Compilers Architecture Synthesis Embedded Syst.*, Nov. 2001, pp. 203–210.
[36] *N2C Training Manual*, CoWare, San Jose, CA, 1999.
[37] *MB8683x User's Guide*, Fujitsu Microelectronics, Inc., Tokyo, Japan.
[38] *SPARClite Series 32-Bit RISC Embedded Processor MB86832 Databook*, Fujitsu Microelectronics, Inc., Tokyo, Japan, 1998.
[39] *1995 DRAM Databook*, IBM, White Plains, NY, 1994.

Accurate Crosstalk Noise Modeling for Early Signal Integrity Analysis

Li Ding, David Blaauw, and Pinaki Mazumder

***Abstract*—In this paper, we propose an accurate and fast method to estimate the crosstalk noise in the presence of multiple aggressor nets for use in physical design automation tools. Since noise estimation is often part of the inner loop of optimization algorithms, very efficient closed-form solutions are needed. Previous approaches model aggressor nets one at a time, assuming that the coupling capacitance to all quiet aggressor nets are grounded. They also model the load from interconnect branches as a lumped capacitor, the value of which is the sum of interconnect and load capacitances of the branch. Finally, previous works typically use simple lumped 2–4-node circuit templates and employ a so-called dominant pole approximation to solve the template circuit. While these approximations allow for very fast analysis, they may result in significant underestimation of the noise. In this paper, we propose a new and more comprehensive fast noise estimation method. We propose a novel reduction technique for modeling quiet aggressor nets based on the concept of coupling point admittance. We also propose a reduction method to replace tree branches with effective capacitors which models the effect of resistive shielding. Furthermore, we model the simplified single aggressor net crosstalk noise problem using a 6-node template circuit and propose a new double pole approach to solve the template circuit. We have tested the proposed method on noise-prone interconnects from an industrial high-performance processor. Our results show a worst case error of 7.8% and an average error of 2.7%, while allowing for very fast analysis.**

***Index Terms*—Crosstalk noise, digital CMOS circuits, interconnect, noise estimation, signal integrity.**

Manuscript received August 15, 2002; revised November 18, 2002. This work was supported in part by the Office of Naval Research, by the National Science Foundation, and in part by the Semiconductor Research Corporation under Contract 2001-HJ-959. This paper was recommended by Associate Editor S. S. Sapatnekar.

The authors are with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109 USA (e-mail: lding@eecs.umich.edu; blaauw@eecs.umich.edu; mazum@eecs.umich.edu).

Digital Object Identifier 10.1109/TCAD.2003.810741