

Power Analysis of Embedded Operating Systems

Robert P. Dick[†], Ganesh Lakshminarayana[‡], Anand Raghunathan[‡], and Niraj K. Jha[†]

[†] Department of Electrical Engineering
Princeton University
Princeton NJ 08544

[‡] CCRL-NEC USA
4 Independence Way
Princeton NJ 08540

Abstract

The increasing complexity and software content of embedded systems has led to the frequent use of system software that helps applications access underlying hardware resources easily and efficiently. In this paper, we analyze the power consumption of real-time operating systems (RTOSs), which 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 characteristics. This work presents the power profiles for a commercial RTOS, $\mu\text{C}/\text{OS}$, running several applications on an embedded system based on the Fujitsu SPARClite processor. Our work demonstrates that the RTOS can consume a significant fraction of the system power and, in addition, impact the power consumed by other software components. We illustrate the ways in which application software can be designed to use the RTOS in a power-efficient manner. We believe that this work is a first step towards establishing a systematic approach to RTOS power modeling and optimization.

1 Introduction

Embedded systems often contain programmable processors and peripherals in addition to application-specific hardware. The complexity of applications and underlying hardware, tight performance/power budgets, as well as aggressive time-to-market schedules, requires the use of run-time software support by application developers. This support usually takes the form of an RTOS, run-time libraries, and device drivers [6, 8, 9, 10, 13, 14, 16]. 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 manages the creation, destruction, and scheduling of tasks, as well as communication between tasks. The device driver and memory management portions of an RTOS simplify the interface between an application and hardware by providing the embedded system designer with routines to manage hardware resources. In addition, an RTOS services synchronous and asynchronous interrupts generated by the processor and other embedded system components.

Typical applications involve significant use of RTOS primitives, the complex interactions of 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. An RTOS accounts for a significant fraction of the computational effort spent by an embed-

ded system. Therefore, designers need to be aware of the potential performance and power impact of their method 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. Our work is a first step towards analyzing and characterizing power consumption in different RTOS components.

Our work focuses on understanding and characterizing the power effects of system software rather than on building a new system-level power analysis tool. To our knowledge, this is the first work that characterizes the power consumption of an RTOS. We demonstrate that the RTOS itself can consume a significant amount of power. The manner in which an RTOS is used significantly affects overall system power consumption. We have developed a general framework to measure the power consumed by different application and RTOS components. We have characterized the power consumption of a commercial RTOS, $\mu\text{C}/\text{OS}$ [9], running on a Fujitsu SPARClite processor. We present quantitative results for energy and time consumed by different operating system tasks, such as context switching, scheduling, inter-process communication, and timer management. We present concrete examples of how information derived from RTOS power analysis can be used to optimize embedded software power consumption. The data and insights derived from our 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.

2 Motivation for RTOS energy analysis

In this section, we illustrate, with examples, the impact of an RTOS on system energy and time consumption. The RTOS is shown to account for a significant fraction of the system's energy consumption. The RTOS energy analysis infrastructure described in Section 3 is used to provide a quantitative breakup of the energy and time consumed by different parts of the application and the RTOS. Our analysis identifies the key sources of energy consumption in the system. Significant savings in energy consumption are obtained by re-writing the application to use the RTOS in a more energy-efficient manner.

Energy consumption information is generally more useful, when optimizing an embedded system's battery lifespan, than power consumption information. 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.

2.1 TCP/IP subsystem example

In our first example, we consider the part of a TCP/IP protocol stack which does checksum computation and interfaces with the Ethernet controller peripheral. Incoming packets are processed to derive their checksums. The packets are subsequently transmitted to the output device (Ethernet controller).

This work was supported in part by a grant from NEC CCRL and in part by the George Van Ness Lothrop Fellowship in Engineering. We would like to thank Dr. Leslie French, from NEC CCRL, for helpful discussions on real-time operating systems and his assistance with the TCP example.

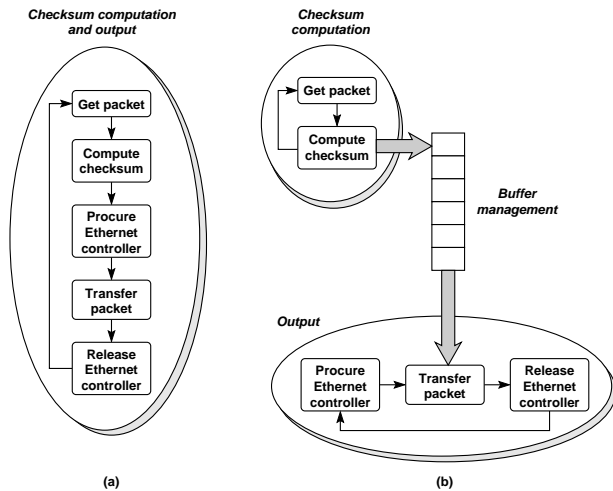


Fig. 1: (a) A straightforward implementation, and (b) a multi-process implementation of the TCP subsystem.

The most straightforward implementation of this algorithm, shown in Fig. 1(a), processes each packet as soon as it is available. However, acquiring a lock on the Ethernet controller’s memory and preparing it to receive a packet, represented by the *Procure Ethernet controller* operation in Fig. 1, may be costly. The *TCP-1* bar in Fig. 4, in Section 4, 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. 1(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, which procures the Ethernet controller and transmits all of the packets in its queue.

The *TCP-2* bar in Fig. 4 shows the energy consumed by the energy-optimized version of the TCP example. This results in a 20.5% overall decrease in energy consumption for the application with most of the savings resulting from reduced reliance on hardware access synchronization and initialization services. Power consumption reduced by 0.2%, *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 the 4.9% increase in energy resulting from the increased complexity of the multiple-task implementation. Note that 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 *TCP-2*. In this example, the RTOS proper consumed only approximately 1% of the overall energy. However, in the other three examples, the RTOS consumes a substantial fraction of the embedded system’s energy.

2.2 Anti-lock braking example

Our second example is based on embedded software used in an automotive anti-lock 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. 2(a) which implements part of an ABS. The system has been adapted from an example in a design automation manual [3]. The *ABS* process calls the *Sense brake pedal* and *Sense speed* functions that sense the brake pedal and the current angular velocity of the wheel, respectively. It 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, which

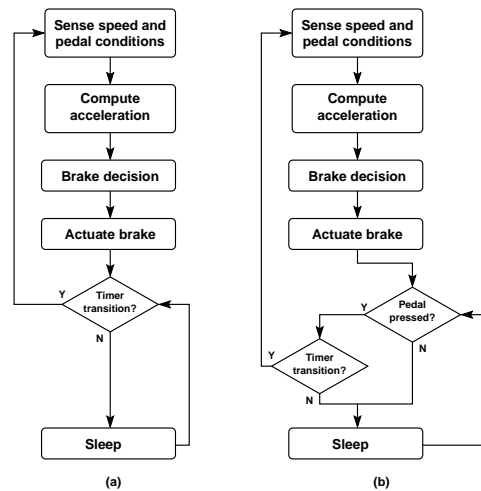


Fig. 2: (a) A straightforward implementation, and (b) an energy optimized implementation of the ABS example.

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 *ABS-1* bar of Fig. 4.

In the straightforward implementation of the ABS example, illustrated in Fig. 2(a), the processor is awakened and the *ABS* process executes with every timer tick. As a result, it frequently wakes up without changing the condition of the brake calipers. This needless execution requires energy which 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. 2(b)), the embedded system’s energy consumption is reduced. As shown in the *ABS-2* bar of Fig. 4, the energy-optimized implementation of the ABS example consumes 62.8% 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 DRAM to remain in 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 seconds, power consumption also reduced by 62.8% in the energy-optimized version. In this example, operating system and board support services accounted for approximately 50% of the system’s energy consumption.

The examples presented in this section demonstrate the manner in which RTOS services are used may have a substantial indirect impact on application energy consumption. Understanding RTOS time and energy effects allows a designer to optimize the energy consumption of the embedded system.

3 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.

3.1 Inputs and outputs

Our framework can analyze the energy consumption of an application, consisting of multiple tasks, executing under a multi-tasking operating system. These tasks interact with each other, as well as with peripheral devices such as UARTs, brake sensors, and other hardware components. The system is simulated to obtain a detailed report of the energy consumed by different application/RTOS functions.

Fig. 3 depicts our energy analysis framework. The application, which consists of multiple processes, is compiled and linked together with the μ C/OS RTOS and Fujitsu’s SPARClite run-time libraries. In addition, a model of the system’s environment or external stimuli is provided to our framework.

The outputs of our tool, shown at the right of Fig. 3, include

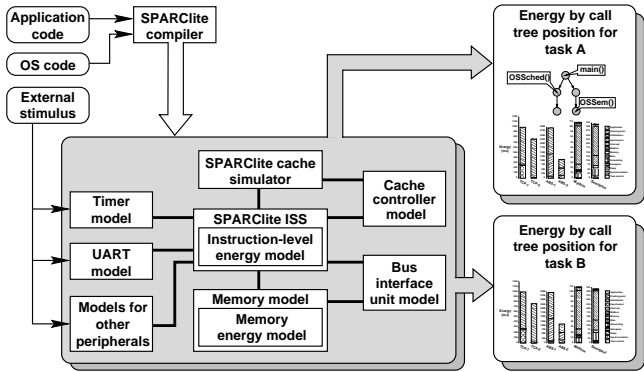


Fig. 3: Energy analysis framework.

call-trees for each task, as well as the RTOS. Each tree node corresponds to a function call, and has a child node for each function call instance which 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.

3.2 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 ASICs and other peripherals. Our energy analysis infrastructure is built around a Fujitsu SPARClike processor, connected to two fast page mode DRAMs, a timer, a UART, and a number of other peripherals. It is easy to add new hardware, *e.g.*, the brake sensors used in the ABS-1 and ABS-2 examples, to the simulated system. 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 $\mu C/O S$. Applications run under $\mu C/O S$.

In order to analyze the energy consumption of the system, we need detailed functional models and energy models for its constituent parts. Instruction level power models for the Fujitsu SPARClike processor and internal cache can be found in the literature [15]. The internal operation of the MB86832 processor is simulated using an instruction set simulator (ISS) [11] which we have enhanced to properly handle interaction with other components in the modeled embedded system. The ISS accurately captures 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, the ISS is enhanced using an on-line cache simulator designed specifically to model the SPARClike processor's cache. It is necessary to use an on-line cache simulator in order to know, during execution, whether or not a miss has occurred. An off-line cache simulator would not allow the correct simulation of an embedded system because, due to races with 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 external memory as well. Specifically, we simulate the cache and on-board bus interface unit of a Fujitsu MB86832 [4, 5], as well as the operation of two IBM0118160PT3-60 low-power fast page mode DRAMs [7].

Memory energy consumption is derived from the manufacturer's data-sheet, and depends on the DRAM's mode of operation. 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 [1, 2, 12].

As mentioned earlier, our energy analysis framework organizes the energy consumption data by function, *i.e.*, the energy consumed in the system during different instances of invocation of a function are combined into a histogram. 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. $\mu C/O S$ performs scheduling and context switches through the function `OSSched`. Our framework uses this information to keep track of context switches. Function calls are performed using the `jmp l` instruction from the SPARC assembly language. The name of the function to which control flow is transferred is determined from the symbol table, which associates an address with each function and global variable. The problem of tracking returns from function calls is complex and requires information specific to (i) the instruction set architecture of the processor being used, (ii) the manner in which the compiler translates different control-flow constructs in the high-level programming language into assembly code, and (iii) some information specific to the RTOS code that performs context switching.

Our energy analysis technique is non-intrusive. This contrasts 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, as explained in Section 3.2, 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 which communicate with the processor. In several systems, even minor 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 four gigabytes of memory, the simulator takes 39.8 minutes to simulate the 14 second ABS-1 example and 12.3 minutes to simulate the 2.5 second TCP-1 example.

3.3 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 $\mu C/O S$ RTOS.

Our system is built around a Fujitsu SPARClike 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, which 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 MB86832 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, which can complete 32-bit multiplications in five cycles. The processor also has a power-down mode, which can be employed to reduce energy consumption.

$\mu C/O S$ is Jean Labrosse's portable real-time kernel for microprocessors and micro-controllers. We use the version Brad Denniston ported to the MB86832 processor. $\mu C/O S$ has been used in many commercial applications, and its performance is comparable to that of other commercial real-time RTOSs. $\mu C/O S$ 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 light-weight version of $\mu C/O S$. The RTOS pro-

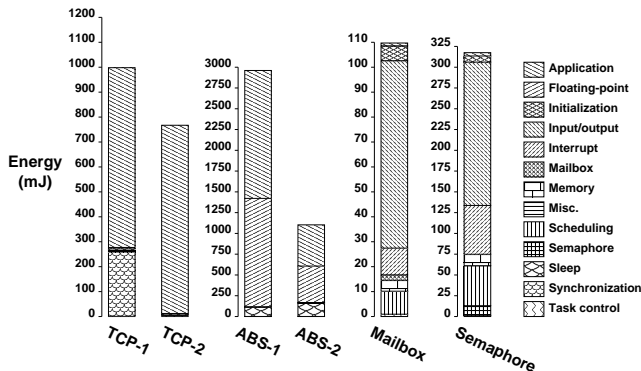


Fig. 4: Energy consumption profiles

vides a number of services such as scheduling, task-management, inter-process communication, memory management, interrupt handling, and timer-related services. We chose μ C/OS for our experiments because it is modular, and its source code is readily accessible. Further information on μ C/OS can be found on the Internet at <http://www.uCOS-II.com>, or in Labrosse's book [9].

4 Results and case studies

We analyzed the energy consumption of μ C/OS RTOS when running several embedded applications. In all cases, we targeted the Fujitsu SPARClite processor based embedded system presented in Section 3.2. Some applications were abstracted from real embedded system application software, while others were designed to exercise specific RTOS functions and services. 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 and application service type. Fig. 4 shows the energy consumed by different RTOS and application services. Each vertical bar represents a distinct example. Vertical bars are divided to indicate the percentage of energy consumed in the various RTOS and application 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.

The Mailbox example illustrates the use of mailboxes for inter-process communication. It consists of three application tasks which communicate via the shared memory mailbox communication service provided by μ C/OS. The tasks also perform writes to the UART. Fig. 4 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 relied heavily on RTOS and processor support routines. As a result, the application code only consumed 1.0% of the total system energy, with RTOS and processor support services consuming the other 99.0%.

The Semaphore example illustrates inter-procedure communication through the use of shared memory. In this case, RTOS primitives which post and release semaphores account for a small but significant portion of the system's energy consumption. The application code consumed 1.3% of the total system energy, with RTOS and processor support services consuming the other 98.7%.

The TCP and ABS examples are described in Section 2. In the ABS-2 example, energy consumption was approximately evenly divided between the SPARClite processor and its DRAM. In the rest of the examples, the processor and DRAM consumed approximately 40% and 60% of the system's energy consumption, respectively. The results in this section, and in Section 2, indicate that an embedded system's RTOS 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 may vary dramatically from approximately 1% (TCP-2) to 99% (Mail-

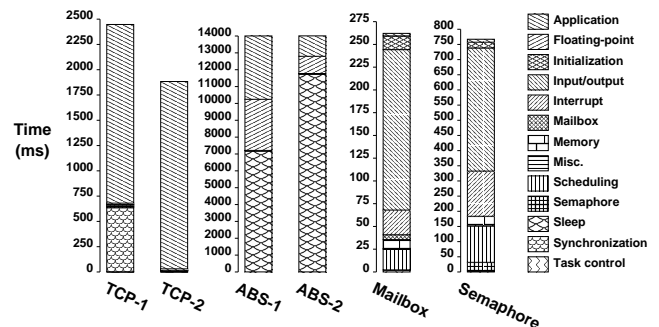


Fig. 5: Time consumption profiles

box), depending on the degree to which the application code 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, and power, consumption by more wisely using RTOS services, as demonstrated in the ABS-1 and ABS-2 examples.

5 Conclusions and future directions

By analyzing a commercial RTOS, μ C/OS, 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. We have presented a method of analyzing the effects of RTOS policies on embedded system power consumption. Insights derived from RTOS power analysis were used to optimize embedded software power consumption. The data and insights derived from our work can be used to drive research on high-level power modeling of different RTOS components. Furthermore, our work enables power-efficient RTOS and application design, and may be incorporated into power-aware system-level design tools.

References

- [1] L. Benini and G. De Micheli, *Dynamic Power Management: Design Techniques and CAD Tools*. Kluwer Academic Publishers, Norwell, MA, 1997.
- [2] A. R. Chandrakasan and R. W. Brodersen, *Low Power Digital CMOS Design*. Kluwer Academic Publishers, Norwell, MA, 1995.
- [3] CoWare, *N2C Training Manual*. 1999.
- [4] Fujitsu Microelectronics, Inc., *MB8683x User's Guide*.
- [5] Fujitsu Microelectronics, Inc., *SPARClite Series 32-Bit RISC Embedded Processor MB86832 Databook*. 1998.
- [6] S. Heath, *Embedded Systems Design*. Butterworth-Heinemann, 1997.
- [7] IBM, *1995 DRAM Databook*. 1994.
- [8] J. J. Labrosse, *Embedded Systems Building Blocks*. R & D Books, Lawrence, KS, 1997.
- [9] J. J. Labrosse, *MicroC/OS-II*. R & D Books, Lawrence, KS, 1998.
- [10] P. A. Laplante, *Real Time Systems Design and Analysis: An Engineers Handbook*. IEEE Press, Piscataway, NJ, 1993.
- [11] Y. Li and J. Henkel, "A framework for estimating and minimizing energy dissipation of embedded HW/SW systems," in *Proc. Design Automation Conf.*, pp. 188-193, June 1998.
- [12] J. Rabaey and M. P. (Editors), *Low Power Design Methodologies*. Kluwer Academic Publishers, Norwell, MA, 1996.
- [13] R. Sharma, "Distributed application development with Inferno," in *Proc. Design Automation Conf.*, pp. 146-150, June 1999.
- [14] D. Stepper, N. Rajan, and D. Hui, "Embedded application design using a real-time OS," in *Proc. Design Automation Conf.*, pp. 151-156, June 1999.
- [15] 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, pp. 437-445, Dec. 1994.
- [16] W. Warner, "Non-pre-emptive multithreading performs embedded software's juggling act," *Electronic Design News*, vol. 44, pp. 117-126, July 1999.