Low Power Operating System for Heterogeneous Wireless Communication Systems

Suet-Fei Li, Roy Sutton and Jan Rabaey
Department of Electrical Engineering and Computer Science
University of California at Berkeley
(suetfei,rsutton,jan)@eecs.berkeley.edu

Abstract

Operating systems in embedded wireless communication increasingly must satisfy a tight set of constraints, such as power and real time performance, on heterogeneous software and hardware architectures. In this domain, it is well understood that traditional general-purpose operating systems are not efficient or in many cases not sufficient. More efficient solutions are obtained with OS's that are developed to exploit the reactive event-driven nature of the domain and have built-in aggressive power management. As proof, we present a comparison between two OS's that target this embedded domain: one that is general-purpose multi-tasking and another that is event-driven. Preliminary results indicate that the event-driven OS achieves an 8x improvement in performance, 2x and 30x improvement in instruction and data memory requirement, and a 12x reduction in power over its general-purpose counterpart. To achieve further efficiency, we propose extensions to the event-driven OS paradigm to support power management at the system behavior, system architecture, and architecture module level. The proposed novel hybrid approach to system power management combines distributed power control with global monitoring.

1. Introduction

The implementation of small, mobile, low-cost, energy conscious devices has created unique challenges for today's designers. The drive for miniaturization and inexpensive fabrication calls for an unprecedented high level of integration and system heterogeneity. Limiting battery lifetimes make energy efficiency a most critical design metric and the real time nature of applications impose strict performance constraints.

To meet these conflicting and unforgiving constraints, we must rethink traditional operating system approaches in embedded wireless communication. General-purpose operating systems developed for broad application are increasingly less suitable for these types of complex real time, power-critical domain specific systems implemented on advanced heterogeneous architectures. The current practice of independently developing the OS and the application, in particular the paradigm of blindly treating a task as a random process, is unlikely to yield efficient implementation [1]. What is needed is an OS that is more intimately coupled to, aware of, and interactive with its managed applications. Specifically, a "lean" but capable OS that is developed to target the nature of these reactive event-driven embedded systems. It should execute with minimal overhead, be agile, and deploy aggressive power management schemes to drive down overall system energy expenditure.

To illustrate these concepts, we construct our argument in two steps. To demonstrate the benefit of a specialized OS that closely matches the application, we will first present a detailed comparison between two OS implementation of the same design -- a wireless protocol stack. The first is eCOS [2], a popular embedded general-purpose multi-tasking OS and the second is an event-driven OS called TinyOS [3]. Preliminary results indicate that the event-driven OS achieves an 8x improvement in performance, 2x and 30x improvement in instruction and data memory requirement, and a 12x reduction in power over its general-purpose counterpart.

The results are certainly very positive, however, we believe that further improvement can be obtained from proper extension of TinyOS. TinyOS possess certain qualities that are very attractive for low power heterogeneous systems. Its event-driven asynchronous characteristics can naturally support the interactions and communications between modules of vastly different behavior and processing speeds in a heterogeneous system. Its simplicity incurs minimal overheads and it has some support for concurrency.

Nevertheless, TinyOS has its own limitations and is insufficient to fulfill the ambitious role demanded by low power heterogeneous systems. First at all, TinyOS primitives are microprocessor centric, while advanced system architectures consist of heterogeneous modules of custom logic, programmable logic, memories, DSPs, embedded processors, and other optimized domain specific modules. Furthermore, TinyOS only supports rudimentary power management scheme. The logical next step is to extend TinyOS and establish it as the global management framework that incorporates the heterogeneous architecture modules in the system, as well as devise sophisticated power management mechanisms.

The rest of the paper is organized as following: Section 2 presents a detailed comparison between two OS implementations of the same wireless protocol design; Section 3 proposes a low power reactive operating system for heterogeneous architectures and the associated global and local power management strategies; and Section 4 concludes the paper.

2. Event-driven versus general-purpose OS

A close "match" between the application and of the OS greatly improves opportunity to efficient final implementation. By match we mean to have Models of Computation (MOC) [4] that are similar to that of the application. MOC is a formal abstraction that defines the interaction of the basic blocks in the system behavior. In particular, three important properties of the specification: sequential behavior, concurrent behavior, and communication have to be clearly defined.

In the following section, we will present a comparison between a traditional general-purpose multi-tasking OS and an event-driven OS in terms of MOC, generality, communication, concurrency support, and memory and performance overhead. The implementation of a wireless protocol design is used as the case study for both.

2.1 PicoRadio II Protocol Design

PicoRadio [4] is an ad hoc, sensor-based wireless network that comprises hundreds of programmable and ultra-low power communicating nodes. PicoRadio applications have the following characteristics: low-data rate, ultra-low power budget, and mostly passive event-driven computation. Reactivity is triggered by external events such as sensor data acquisition, transceiver I/O, timer expiration, and other environmental occurrences. The chosen MOC for the PicoRadio protocol stack is Concurrent Extended Finite State Machines (CEFSM) [5]. CEFSM models a network of communicating extended finite state machines (EFSM), which are finite state machines that effectively express both control and the computation found in datapath operations. Each layer in the protocol stack is modeled as an EFSM. The communication between EFSMs is asynchronous because the stack layers work at differing rates: the lower layers typically run much faster than the higher layers.

In this second version of the PicoRadio design (PicoRadio II), the protocol stack has a simple User Interface (UI) layer, a transport layer, a MAC layer, and an interface to the physical layer. Different layers in the stack have vastly different processing granularities and speeds: Physical layer processes at bit level and has to respond in microseconds, while UI reacts to user requests in seconds and even minutes. Due to their different behavior activity and characteristics, the UI and transport layers are to be implemented in software on the embedded processor while the MAC and physical layers are implemented with the support of custom optimizes hardware modules.

2.2 General-purpose Multi-tasking OS

The general-purpose multi-tasking OS was originally developed for the PC platform and later adapted for general embedded systems. It is good for supporting several mostly independent applications running in virtual concurrency. Suspending and resuming amongst the processes when appropriate provide support for multi-tasking and/or multi-threading. Inter-task communication involves context switching which can become an expensive overhead with increased switching frequency. This overhead is tolerable for PC applications since the communication and hence switching frequency is typically low when compared to the computation block granularity. Moreover, as these overheads grow, the wasted energy expenditures are of relatively little concern for these virtually infinite energy systems. As general-purpose OSs do not target low power applications, they have no built-in energy management mechanisms and any employed are wholly deferred to the application with its limited system scope.

It is apparent that the MOC of the general-purpose OS is quite different from that of the protocol stack. The processes across the layered protocol stack are not independent. They are coupled and activate and deactivated with events from neighboring processes. In other words, the communication frequency is high amongst neighbors and high overheads are far less tolerable. As we will see shortly, this MOC "mismatch" results in major inefficiencies.

We have designed a chip to implement the PicoRadio II protocol stack. Our main design tool is Virtual Component Codesign (VCC) from Cadence Design Systems [6]. The VCC flow covers the entire design process from behavior specification to architecture exploration, all the way down to final hardware/software implementation. The software implementation process in VCC takes a traditional approach and assumes a general purpose multi-tasking OS. The code generation is accomplished by turning each concurrent system component in the specification into a task. Communication wrapper functions are then generated to connect the tasks and the OS. We have chosen the popular eCOS as our embedded OS due to its availability and efficiency.

From the chip layout, we notice that the software portion of the architecture including the processor and its memory blocks occupies more than 70% of the total area. This is especially inefficient considering that the processor is greatly under-utilized (utilization < 7%). Reason being that the software-implemented UI and transport layers run at much lower activity and rate (user request and packet level processing) than the hardware-implemented MAC and physical layers (bit level processing).

Careful analysis of the software code reveals that of the total 10K byte instruction code size, about 50% is communication overhead. The massive data memory size of 54K is a result of the communication overhead, expensive scheduler overhead, memory management, and stack allocations.

2.3 Event-driven OS

TinyOS specifically targets event-driven communication systems. Its MOC is CEFSM, which matches that of the protocol processing system. This match drastically reduces the communication overhead as well as other OS related costs. Because TinyOS is not designed to support a broad range of general applications, it can cut down on expensive OS services such as dynamic memory allocation, virtual memory, etc. In addition, unnecessary performance-degrading polling is eliminated and context switching is minimized and very efficiently implemented.

In TinyOS, an application is written as a graph of components. For the PicoRadio II example, components would be the layers in the protocol stack. Each component has command and event handlers that process commands and events from other components, tasks that provide a mechanism for threaded description, and a static frame that stores internal state and local variables.

The TinyOS system operation can be briefly described as following: external events from the RF transceivers or sensors propagate from the lowest layers up the component graph until handled by the higher layers. To prevent event loss, the system must process incoming events faster than their arrival rate. Threaded behavioral description is supported via tasks, which are operations in the event or command handlers that require a "significant" number of processor cycles. Tasks are pre-empted by the arrival of an incoming event and are dispatched from a task queue. TinyOS uses a simple FIFO task scheduler. Built-in power control is exercised by shutting down the CPU when no tasks are present in the system after all event processing.

We have re-implemented the PicoRadio II protocol stack using TinyOS. In the next section, we will present a comparison between the general-purpose OS (eCOS) and the event-driven OS (TinyOS) in three important performance metrics: memory requirement, performance, and power.

	General purpose OS	Event-driven OS
MOC	Multi-tasking	Communicating EFSMs
Generality	General	Target event driven systems
Communication Overhead	Large	Small
Communication Frequency	Infrequent	Frequent
Memory Requirement	Large	Small

Table 1: General comparison.

OS type	Processor	Application	Total instruction mem	Data mem
General Purpose	ARM7 thumb	**10,096**	**22324**	**54988**
Event-driven	ARM7 thumb	**5312**	**8000**	**2800**
Event-driven	8 bit RISC	2740	3176	709

Table 2: Memory requirements comparison.

2.4 Comparison Summary

Table 1 summarizes the contrast between the two OS's as presented in Section 2.2 and 2.3. Table 2 shows the memory requirement comparison between the two OS's. With the same processor selection (16 bit ARM7), TinyOS needs half the instruction memory and one-thirtieth the data memory. Studies showed that the power consumption of SRAM scales roughly as the square root of the capacity [7]. This implies that with TinyOS, instruction memory power can be reduced by 1.6x, and data memory power by 4.2x. Using a simpler processor such as 8-bit RISC could further reduce memory size and power consumption.

Figure 1 presents the performance comparison. The left graph compares the total processor cycle count: 16365 vs. 2554. TinyOS shows a factor of eight improvements, which translates directly to a factor of eight reductions in processor power consumption. The right graph compares the OS overhead (the lowest portion of the bars) as a percentage of the total cycles. As an indication of its inefficiency, the general-purpose OS has an

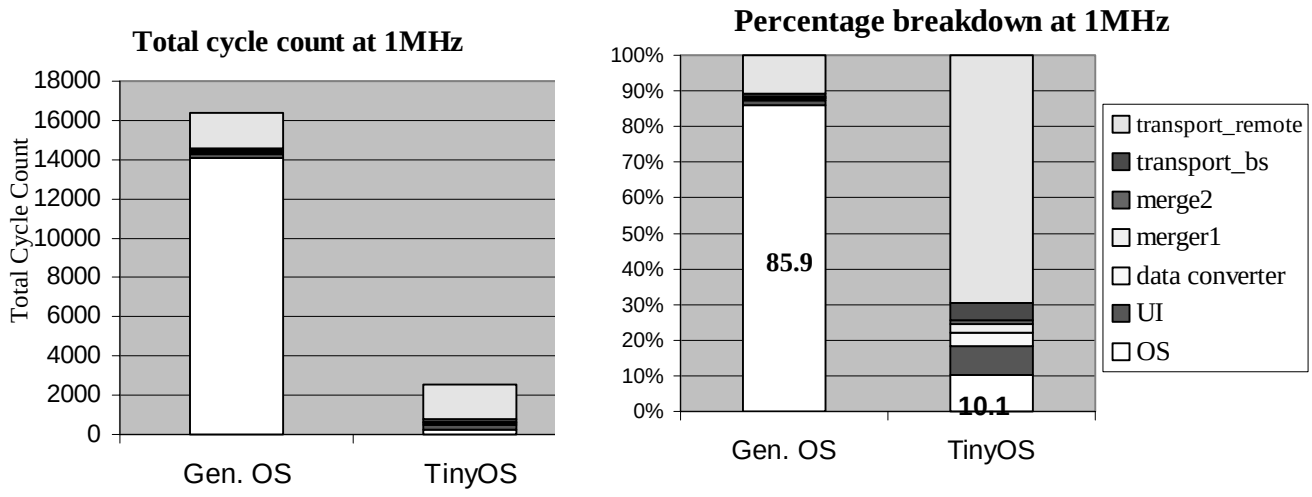

Figure 1: General-purpose versus event-driven OS. Key at right identifies system components.

OS overhead of 86% while TinyOS has 10%.

Now let us calculate how much power is actually saved considering both the processor and its memory blocks. With a 0.18µm technology and a supply voltage of 1.8V, an ARM7 consumes 0.25mW/MHz. For a memory size of 64KB, read per access consumes 0.407mW/MHz and write consumes 0.447mW/MHz. Assume that 10% of the instructions involve memory read operations and 10% memory writes and apply memory size as well as processor cycle count scaling, the power consumption for the two OSs are: 0.608mW/MHz and 0.053Mw/MHz. That is, TinyOS demonstrates a factor of 12 improvements in power.

One may argue that eCOS is an overkill and could be optimized to yield better performance and power. ECOS is a reconfigurable OS, and the authors choose the configuration options that yield the smallest and simplest implementation. While further optimization could be applied, with a factor of 12 win in power, TinyOS should remain far superior.

3. Low Power Reactive OS for Heterogeneous Architectures

Our driving research goal is to design an energy efficient OS for domain specific heterogeneous architectures. We believe that some basic TinyOS concepts are very attractive and can be adopted to reach such a goal. TinyOS's event-driven asynchronous characteristics can naturally support the interactions and communications between modules of vastly different behavior and processing speeds in a heterogeneous system. Its simplicity reduces overheads and leads to more power efficient implementation. It also provides some support for multiple flows of control that are typical of wireless sensor applications.

However, TinyOS has its limitations and is insufficient for our research goal. It has to be properly extended to the system level to include management of not only computation on the embedded processor, but also computation on the optimized architecture modules. In the following sections, we will elaborate on the roles of the "system level OS" in the context of PicoRadio III, a next generation heterogeneous wireless communication system and discuss the necessary TinyOS extensions.

3.1 Event-driven Global Scheduler and Power Management

In a complex heterogeneous system, the OS acts like a hardware abstraction layer [8] that manages a variety of system resources. For a power critical application, simplicity should be the primary design philosophy. The OS should perform a dedicated set of indispensable duties and only these duties. There are two basic OS duties: concurrency management at both application and architecture levels and global power management.

Wireless sensor applications typically have multiple flows of control and data. A sensor node can sense the environment, forwards packets and receive commands all at the same time. The OS needs to support concurrency in the application as well as explore and utilize the concurrency in the heterogeneous architecture. Since the OS has the global "view" of the system, it can also perform global power management to optimize the system power consumption. Essentially, the OS is a global scheduler with power management.

In our vision of the system architecture, the OS is refocused from the microprocessor and becomes a separate unit that can be implemented in software, hardware, reconfigurable logic, or some combination. Traditional software centric control approaches have a central control unit that schedules communication between the system modules. We believe that a more efficient approach is to distribute control over the entire system. Global monitoring is added to further improve the system performance via feedback derived from observations with greater system scope. In this control framework, communication can occur between certain modules without the intervention of the OS. When dependent reactive behaviors are mapped to interconnected architecture modules, their communication can occur through the established hardware event channels.

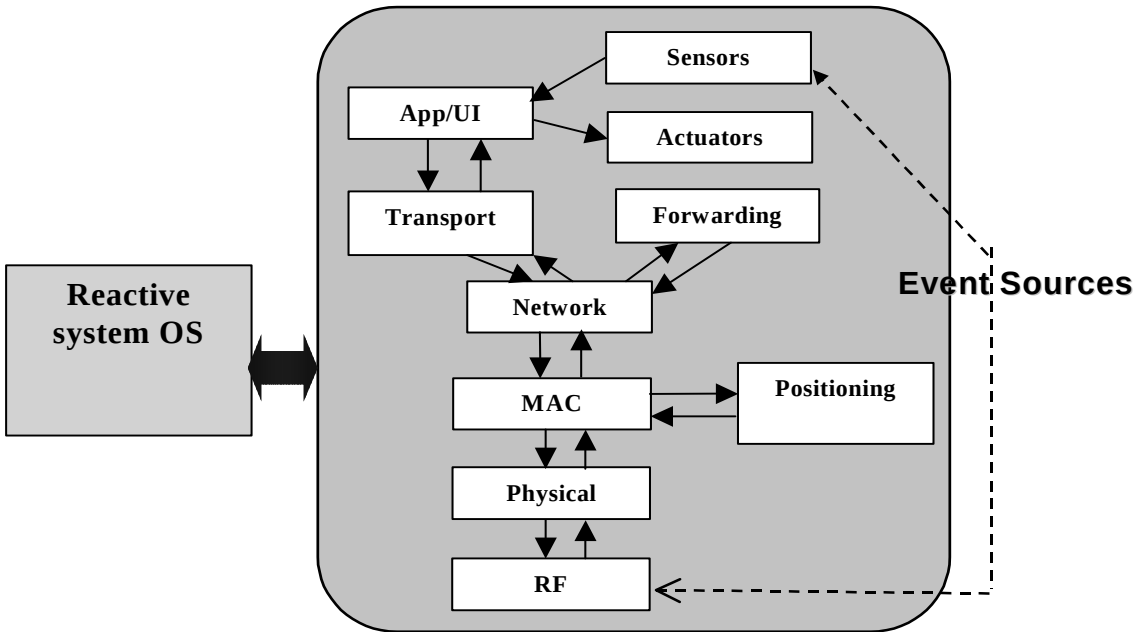

Figure 2: Behavior diagram of the PicoRadio sensor node.

Power management should be applied at all levels of the design hierarchy: System level, architecture module level, circuit level and device level [4]. By carefully incorporating local power management into the individual architectural modules, we can push power management down the design hierarchy. We propose a hybrid approach to power management, which combines distributed power controls with global monitoring.

PicoRadio networks are sensor networks characterized by bursty and mostly aperiodic traffic. The low-duty cycle makes it essential that individual modules are powered down whenever not active. If not, leakage current will dominate the power budget. Rather than assuming all the modules in the systems are on and could be turned-off to conserve energy, we assume that they are "off" until powered-up by the arrival of events at their interface. Internally, modules are awakened either by their neighbors or by the OS. This novel approach assumes the concept of a wake-up radio, which only turns on when communications are truly needed [9].

Figure 2 is the behavior diagram of the PicoRadio III sensor node. It shows the different components in the system and the interactions between them. Communication between components is purely reactive. Figure 3 shows Architectural diagram for PicoRadio sensor node System. Each module has an interface that is responsible for its own local power status and control. When an event arrives from a neighbor, the interface logic will decide which part of the

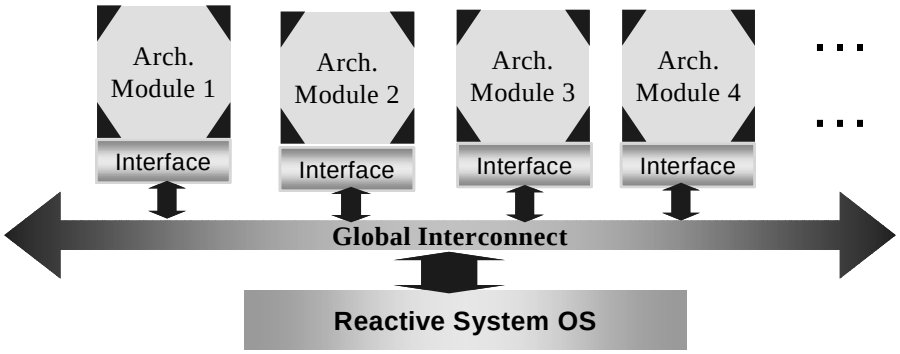

Figure 3: Architectural diagram for PicoRadio sensor node.

module should be turned-on to process the event. The interface could also have voltage-scaling capability built-in to further control the power consumption of the module by matching module performance, and hence energy expenditure, with workload. The sleeping mechanism can be implemented as some function of the module idle time and wake-up overhead, etc.

On top of this distributed power control mechanism, global monitoring is implemented to incorporate global information that local modules are not able to "see". For this, the system level OS supports global power management. It maintains global state by monitoring the module interactions, and schedules periodic system maintenance accordingly. Based on its knowledge of the entire system, it issues commands to a module's interface to override local decisions when there are conflicts of interest. For example, the network layer may wish to go to sleep since it has not received any event for a certain amount of time, but the OS senses some activity in the RF layer and might find it advantageous to prevent the network layer from going to sleep.

3.2 TinyOS Limitations and proposed extensions

Given our revised, broader prospective of the operating system, TinyOS is limited and inadequate. It is primarily designed for uni-processor architectures. All components except for the lowest layers of the application are implemented in software. Low-level hardware components are required to have a software wrapper to interact with the scheduler and the rest of the system. This software centric approach does not allow full exploration of the integrated, heterogeneous system architecture. Moreover, TinyOS assumes off the shelf components and in essence has no access to customized power-efficient blocks. TinyOS's rudimentary power management scheme also needs to be greatly improved.

We have proposed the following extensions to TinyOS to establish the OS as the global system scheduler and power manager.

1. Replace the simple FIFO task scheduler in TinyOS with a more sophisticated scheduler, which supports voltage scaling of the modules to which the tasks are assigned. This implies that each task should carry some real time scheduling information. Scheduling techniques for variable voltage can be applied to minimize power consumption while meeting the performance constraints [10] [11][12].
2. Components can also be implemented in hardware. Moreover, hardware components need not dispatch tasks. Tasks are introduced in TinyOS to implement threads in the algorithm on uni- processor architectures. While executing a task, the processor can be pre-empted to handle higher priority incoming events. If components are implemented in hardware, tasks are no longer needed and are removed for simplicity.
3. Tasks are dispatched to either software or hardware. This is to best utilize the whole system resources since dedicated architectural modules could be designed to support specific tasks. (In TinyOS, all tasks go into software.)
4. Add event queues at the lowest layers. This can reduce the external event losses and make the system more robust. The current TinyOS has no queue implementation.
5. Add global power control mechanisms. The OS should collect runtime profiles and statistics, perform periodic system maintenance operations and maintain system level power state.

4. Conclusion and Future Work

In this paper, we have presented issues concerning the implementation of a low power operating system for heterogeneous communication systems. We argue that the OS should have a MOC that closely matches the application, and showed the significant improvement of the event-driven TinyOS over a popular general-purpose OS as a proof-of-concept. We have also discussed necessary extension to TinyOS for supporting heterogeneous architectures, and proposed a novel system power management framework. The next step is to build a simulation environment to fine tune the concepts, and eventually implement the OS on the PicoRadio III system.

5. Reference

[1] K.Ramamritham and J.A. Stankovic, "Scheduling Algorithms and Operating Systems Support for Real-Time Systems",

Proceedings of the IEEE, January 1994, pp. 55-67.

[2] http://sources.redhat.com/ecos

[3] David Culler et al, *The TinyOS group*, Department of EECS, UC Berkeley.

[4] J. Rabaey et al., "PicoRadio Supports Ad Hoc Ultra-Low Power Wireless Networking", *IEEE Computer*, Vol. 33, No. 7, July 2000, pp. 42-48.

[5] E. Lee and A. Sangiovanni-Vincentelli, "A Unified Framework for Comparing Models of Computation", *IEEE Trans. on Computer Aided Design of Integrated Circuits and Systems*, Vol. 17, N. 12, pp. 1217-1229, December 1998.

[6] http://www.cadence.com

[7] R. Evans & P. Franzon, "Energy Consumption Modeling and Optimization for SRAM's", Journal of Solid-State Circuits, Vol. 30, No. 5, May 1995.

[8] A. Ferrari and A. Sangiovanni-Vincentelli, "System Design: Traditional Concepts and New Paradigms", *Proceedings of the Int. Conf. on Computer Design*, Austin, Oct. 1999.

[9] B. Otis, I. Telliez and I. Cambonie, "PicoRadio RF", http://bwrc.eecs.berkeley.edu/Local/Research/PicoRadio/PHY, January 2001.

[10] Y Lin, C. Hwang & A. Wu, "Scheduling Techniques for Variable Voltage Low Power Designs", *ACM Transaction on Design Automation of Electronic Systems*, Vol. 2, No. 2, April 1997, pp 81-97.

[11] J. Monteiro, S. Devadas, P. Ashar, A. Mauskar, **"**Scheduling techniques to enable power management"**,** *33rd Design Automation Conference*, June 1996.

[12] J. Brown, D. Chen, G.W. Greenwood, Xiaobo Hu; R. Taylor, "Scheduling for power reduction in a real-time system", *Proceedings 1997 International Symposium on Low Power Electronics and Design*.