Rate-Harmonized Scheduling for Saving Energy

Anthony Rowe Karthik Lakshmanan Haifeng Zhu† Ragunathan (Raj) Rajkumar
Dept. of Electrical & Computer Engineering
Carnegie Mellon University, U.S.A.
{agr, klakshma, haifengz, raj}@ece.cmu.edu

***Abstract*—Energy consumption continues to be a major concern in multiple application domains including power-hungry data centers, portable and wearable devices, mobile communication devices and wireless sensor networks. While energy-constrained, many such applications must meet timing and QoS constraints for sensing, actuation or multimedia data processing. Many modern power-aware processors and microcontrollers have built-in support for active, idle and sleep operating modes. In sleep mode, substantially more energy savings can be obtained but it requires a significant amount of time to switch into and out of that mode. Hence, a significant amount of energy is lost due to idle gaps between executing tasks that are shorter than the required time for the processor to enter the sleep mode. We present a technique called *Rate-Harmonized Scheduling* that naturally clusters task execution such that processor idle times are lumped together. We next introduce the *Energy-Saving Rate-Harmonized Scheduler* which guarantees that *every* idle duration on the processor can be used to put the processor into sleep mode. This property can be used to even eliminate the idle power mode in processors but nevertheless it is predictable, analyzable, and saves more energy. We finally evaluate the practical benefits of Rate-Harmonized Scheduling implemented in the nano-RK real-time operating system [1] for wireless sensor networks.**

I. Introduction

The functionality and continual use of increasingly popular portable, mobile and wearable communication devices are often significantly constrained by the limits of their energy sources. Extending battery lifetime is a major challenge in wireless sensor networks. At the other end of the size scale, the energy demands of running and cooling data centers are making them very expensive. Hence, the need to extract energy savings continues to garner attention among multiple communities.

We propose, analyze and demonstrate the benefits of some simple and practical yet effective algorithms for saving energy. These schemes can be readily implemented in reservation-based real-time operating systems (such as Linux/RK [2] and nano-RK [1]), where the processing, bandwidth and timing constraints of tasks are known *a priori*.

Most modern micro-controllers have built-in support for various energy saving modes. Typically, there is a longer transition time associated with moving to lower energy states due to the overhead required for the main oscillator to startup and stabilize. On FireFly sensor nodes [3] using the Atmel ATmega1281 processor, the transition from an active energy state to an idle energy state takes on the order of a few micro-seconds since the main system clock remains active. However, the round-trip transition from idle to deep-sleep takes on the order of 10-15ms. If the gap between two tasks is less than this period, the processor is only able to transition to the idle energy state even though there is no useful work to be done. In fact, we have observed that a significant percentage of time is spent in idle mode due to the accumulation of small gaps between tasks.

We introduce a family of rate-harmonized schedulers (RHS) that clusters the execution of tasks so that idle durations can be lumped together enabling transitions to the sleep mode. One such rate-harmonized scheduling technique called *Energy-Saving RHS* adds a virtual sleep task in a manner that allows every inactive period of execution to be used as sleep time in the system. This scheme yields many major benefits:

- A processor using energy-saving RHS can transition *any and every* idle duration on the processor into the sleep mode. The idle slots in the schedule can also be due to a task executing less than its worst-case execution time. Energy saving is thus maximal. The only requirement is that the taskset be feasible under energy-saving RHS, and exact conditions for feasibility are provided.
- Thanks to the maximal energy savings obtained, there is no longer a need to manage more than two CPU energy states. The processor is either in the sleep state or the active state. This simplifies both the scheduler and potentially the hardware design of the processor.
- The worst-case energy consumption using energy-saving rate-harmonized scheduling can be predicted, analyzed and optimized. The benefits of analyzability are likely to manifest themselves over time in myriad and surprising ways.

Rate-harmonized schedulers have the interesting property of clustering task execution together. Though beyond the scope of this work, this batching property can be useful in other scenarios. For example in a situation where mul-

†Author currently at United Technologies Research Center.

tiple tasks access a shared resource that has a significant setup cost, it would be ideal avoid repeated initializations. In this paper, the shared resource we focus on is the CPU and the penalty that we minimize is the energy lost in the setup time required for the CPU to transition from the deep sleep to the idle energy state.

A. Organization of Paper

The rest of this paper is organized as follows. Section II discusses related work. Section III introduces Rate-Haromonizing Schedulers providing schedulability conditions and runtime properties. Section IV describes the FireFly sensor network hardware, the Nano-RK operating system along with a performance evaluation of our energy schemes. Section V provides concluding remarks.

II. Related Work

Many current sensor networking systems are designed using non-preemptive operating systems in order to save on memory [4], [5]. These systems are event-triggered and typically provide energy savings by executing tasks as quickly as possible and then returning to sleep. Without deadline information, it is difficult to cluster events in order to further save energy. Due to the increasing complexity of sensor networking tasks and the scaling of technology, multiple preemptive operating systems capable of running on micro-controllers are now publicly available [6], [7], [1] . These operating systems mention use of *apriori* task knowledge for energy savings, but do not provide schemes with additional benefits beyond standard priority-based scheduling.

For time-sensitive applications, we use priority-based preemptive scheduling to implement the rate-monotonic paradigm [8] of real-time scheduling. Given a periodic sensor task set with timing deadlines and a priority set inversely proportional to the period of the task, one can prove timing guarantees are honored. We extend upon this paradigm through the use of phase adjustment at the cost of scheduling efficiency to improve energy performance. [9] generalizes the original rate-monotonic analysis to support periodic tasks with arbitrary deadlines. We use parts of this analysis to prove utilization bounds of our task sets given phase adjustments due to harmonization.

Real-time scheduling experts will rightfully note that the "energy-saver" task used in energy-saving rate-harmonized scheduling behaves like a sporadic task [10] which executes at the highest priority from a scheduling perspective and services an endless queue of sleep requests. However, when used with basic rate-harmonized scheduling, the technique exhibits an additional set of very attractive properties from an energy-saving perspective.

The techniques of dynamic voltage scaling (DVS) and dynamic frequency scaling (DFS) have been the focus of much research in recent years with the objective of reducing energy consumption. This is due to the fact that the dynamic power consumption of CMOS circuits [11], [12] is given by $P = aC_L V_{DD}^2$, where P is the power, a is the average activity factor, C_L is the average load capacitance, V_{DD}^2 is the supply voltage and f is the operating frequency. Since the power has a quadratic dependency on the supply voltage, scaling the voltage down is an effective way to minimize energy consumption. However, lowering the supply voltage can also adversely affect the system performance due to increasing delay. Many lower cost microcontrollers do not have DVS/DFS capabilities. Our approach works even in systems without support for DVS and DFS. Nevertheless, integrating DVS/DFS techniques into our framework will be an interesting area of future work.

III. Rate-Harmonized Scheduling and Maximal Energy Saving

Most modern processors have built-in support for multiple modes of operation with each mode consuming a different amount of energy. The processor also needs more or less time to switch into and out of different power-saving modes. The lower the power consumption, the larger is the time required to switch into and out of that mode. For example, a "power-aware" processor normally has

- an *active* mode, wherein the processor consumes the most amount of energy E_{active} but it can execute tasks waiting to be processed,
- an *idle* (or *nap* mode), where it consumes less energy E_{idle} than the active mode, but no processing can take place and a small amount of time ST_{idle}[1] must be spent to switch into and out of this mode, and
- a *deep sleep* (or *sleep* mode), where it consumes the least amount of energy E_{sleep}, but no processing can take place and a sizeable amount of time ST_{sleep} must be spent to switch into and out of the mode. This typically involves spinning down the main oscillator which takes a significant amount of time to stabalize upon reactivation.

[1] *ST* stands for *switching time*.

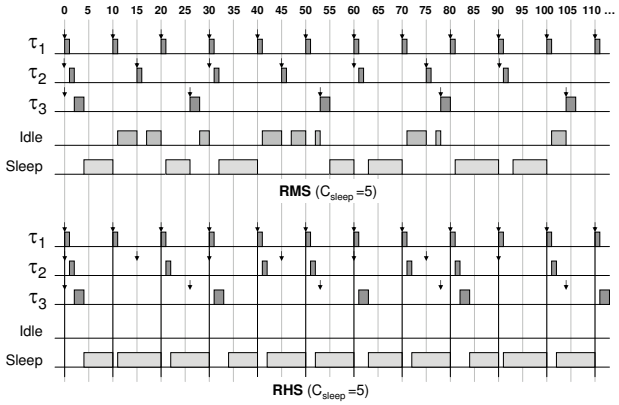

Fig. 1. This figure shows the following task set $\tau_1 = \{1, 10\}$, $\tau_2 = \{1, 15\}$, $\tau_3 = \{2, 26\}$ with a $C_{sleep} = 5$ being scheduled with normal RMS on the top and with RHS on the bottom. In the bottom timeline, the tasks are harmonized to 10 ($\mathbf{T_H} = 10$) which is denoted by the darker vertical bars.

Normally, $E_{active} > E_{idle} >> E_{sleep}$, and $ST_{idle} << ST_{sleep}$.

When the processor has no ready tasks to execute, it is often tempting to switch the processor into the sleep mode. However, the processor cannot go to sleep if a job can arrive within ST_{sleep} units of time. This can result in significant time intervals when the processor is not doing any useful work but is yet not in the deep-sleep mode.

Fortunately, the periodic nature of real-time tasks, when appropriately structured, provide significant insight into the future arrival times of jobs for the processor. Real-time operating systems using a reservation-based approach (e.g. Linux/RK [2] and Nano-RK [1]) can exploit this knowledge to switch the processor into low-power deep-sleep mode of operation, whenever jobs are not expected to arrive in the near future. Thereby, energy savings can be obtained without compromising the timeliness and QoS constraints of application tasks.

We now introduce a novel but simple, practical and effective new technique that maximizes the percentage of time a processor spends in the deep-sleep mode without violating the timing constraints of the real-time taskset. In fact, using our technique, *every* time-unit that the processor is not in active mode can be spent in deep-sleep mode. Hence, the idle-mode of processors can even be potentially eliminated yielding hardware savings along with enhanced energy savings!

A. Notation and Terminology

We will consider a periodic taskset $\{\tau_1, \tau_2, \cdots, \tau_n\}$ comprising of n periodic tasks, each with a worst-case computation time, C_i, and period, T_i. The taskset is ordered such that $T_1 \leq T_2 \leq \cdots \leq T_n$. The relative deadline D_i of each task τ_i is the same as its period T_i. Each task τ_i also has an initial arrival time of ϕ_i, such that its arrival times are at ϕ_i, $\phi_i + T_i$, $\phi_i + 2T_i$, $\cdots$. Without loss of generality, we assume that the phase of task τ_1, $\phi_1 = 0$. We adopt the fixed-priority preemptive scheduling approach with task priorities assigned using the rate-monotonic policy (i.e. inversely proportional to task periods)[2]. The *utilization* U_i of task (τ_i) is given by $\frac{C_i}{T_i}$. The total utilization U_{tot} of the taskset is the sum of the utilization of the tasks in the taskset.

Our proposed approach called *Rate-Harmonized Scheduling* allows the execution of different real-time tasks to be clustered together, thereby having the effect of lumping together idle durations in the processor schedule. While many variants of rate-harmonized scheduling are possible, we introduce only a basic version upon which our maximal energy-saving scheme can be built.

B. Basic Rate-Harmonized Scheduler

A *general* rate-harmonized scheduler (RHS) utilizes a set of periodic values $\{\mathbf{T_H^1}, \cdots, ..., \mathbf{T_H^n}\}$ where $\mathbf{T_H^i} \leq T_i$, $i = 1$ to n, and the values in $\{\mathbf{T_H^1}, \cdots, ..., \mathbf{T_H^n}\}$ are harmonic. These harmonic periods are referred to as the *Harmonizing Base Periods*, and all have the same initial phasing of $\tau_1 = \phi_1 = 0$. Tasks in the given taskset $\tau_1, \tau_2, \cdots, \tau_n$ are released according to their arrival patterns as in the classical periodic taskset model. However, each job of a task τ_i only becomes eligible to execute at its next nearest periodic boundary of $\mathbf{T_H^i}$. Specifically, the $(k+1)^{th}$ job of τ_i arrives at time $\phi_i + kT_i$ but becomes eligible to execute only at time $(p\mathbf{T_H^i} \mid (p-1)\mathbf{T_H^i} < \phi_i + kT_i \ \wedge \ p\mathbf{T_H^i} \geq \phi_i + kT_i)$.

In the *Basic Rate-Harmonized Scheduler*, $\mathbf{T_H^1} = \mathbf{T_H^2} = \cdots = \mathbf{T_H^n} = \mathbf{T_H}$. We refer to $\mathbf{T_H}$ simply as the *Harmonizing Period*. Since this is a rate-harmonized scheduler, we must have $\mathbf{T_H} \leq T_1$. Figure 1 shows an example taskset being scheduled with normal rate-monotonic scheduling (RMS) and Basic RHS with $\mathbf{T_H} = 10$ and $ST_{sleep} = 5$. It assumes that $\phi_1 = \phi_2 = \cdots = \phi_n = 0$. The arrival time of each task is indicated with an arrow above each

[2]Our rate-harmonized scheduling approach can be easily adapted to dynamic priority approaches such as earliest-deadline-first (EDF) scheduling, but it is beyond the scope of this paper.

timeline. In the Basic RHS schedule, tasks that arrive before or after integral multiples of $\mathbf{T_H}$ are *not* eligible to execute until the next closest boundary of $\mathbf{T_H}$ when they are serviced based on their priority. Tasks that are not eligible are delayed until the next $\mathbf{T_H}$ boundary.

$\mathbf{T_H}$ is chosen so as to improve schedulability. Suppose $\Psi = \{\tau_j \mid T_j < 2T_1, j \neq 1\}$. If $\Psi = \emptyset$, $\mathbf{T_H} = T_1$. Otherwise, $\mathbf{T_H} = \frac{T_1}{2}$.

We now prove some properties of basic rate-harmonized scheduling.

Theorem 1. A critical instant for any task τ under basic rate-harmonized scheduling occurs when τ is requested simultaneously with requests for all higher priority tasks, and τ has to wait $\mathbf{T_H} - \epsilon$ before it becomes eligible to execute (where ϵ is an infinitisimally small positive value).

Proof: Under basic rate-harmonized scheduling, all tasks become eligible to execute only at boundaries that are integral multiples of $\mathbf{T_H}$. If τ arrives at t simultaneously with all higher priority tasks $\mathbf{T_H} - \epsilon$ time-units before the next integral boundary of $\mathbf{T_H}$, task τ and all its higher priority tasks become eligible to execute only at $t+\mathbf{T_H}-\epsilon$.

If any higher priority task τ_h arrived earlier than t, τ_h would have been eligible to execute earlier, and the response time for τ cannot become worse. If any higher priority task τ_h arrived after t and at or before $t+\mathbf{T_H}$, all of τ_h's jobs arriving later will be delayed, and the response time for τ can only become better (or stay the same). If τ arrives later than t, its response time will become longer by letting τ arrive at t. If τ arrives earlier than t, it would become eligible to execute earlier and its response time can only become shorter (or stay the same). Hence, t represents a critical instant for τ. ■

Remark: Note that in Theorem 1, relative to the classical Liu and Layland model, the additional delay of $\mathbf{T_H}$ encountered by a task τ is *concurrent* with respect to the delays encountered by all its higher priority tasks.

Lemma 2. The worst-case response time of τ_1 under rate-harmonized scheduling is given by C_1.

Proof: The rate-harmonized scheduler requires that the phasing of the harmonizing period $\mathbf{T_H} = \mathbf{T_H^1}$ be the same as that of τ_1, and $\mathbf{T_H}$ is harmonic with respect to τ_1. As a result, τ_1 is eligible to execute as soon as it arrives. Being also the highest priority task, τ_1's worst-case response time is C_1. ■

Lemma 3. The maximum value of $\frac{\mathbf{T_H}}{T_i} = 0.5$ for any task τ_i, $i \neq 1$.

Proof: The lemma follows from the choice of $\mathbf{T_H}$ for the basic rate-harmonized scheduler. ■

Theorem 4. A taskset is feasible under basic rate-harmonized scheduling if $\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 0.5$.

Proof: By assumption, deadline D_i = period T_i for every task τ_i. From Theorem 1, the additional delay of $\mathbf{T_H}$ encountered by a task τ_i is equivalent to shortening the deadline of τ_i in the Liu and Layland model by $\mathbf{T_H}$ (with the exception of τ_1 from the proof of Lemma 2).

Hence, the ratio of the effective deadline to the period T_i (denoted by Δ_i in [9]) of τ_i, $i \neq 1$, is given by $\Delta = \frac{T_i - \mathbf{T_H}}{T_i}$. From Lemma 2, task τ_1 cannot constitute the bottleneck task. From Lemma 3, the maximum value of Δ is 0.5. The theorem follows Theorem 1 from [9]. ■

An exact schedulability condition can also be stated using the fixed-point response time computation technique [13], [14]. To find the worst-case response time of τ_i, let

$$W_0 = C_i + \mathbf{T_H}$$

Iterate on k as follows:

$$W_{k+1} = C_i + \mathbf{T_H} + \sum_{j=1}^{i-1} \lceil \frac{W_k}{T_j} \rceil C_j$$

until $W_{k+1} = W_k$ in which case, W_{k+1} is the worst-case response time of τ_i. Check if $W_{k+1} \leq D_i$. If $W_{k+1} > D_i$, τ_i misses its deadline and the computation can be stopped.

C. Energy-Saving Rate-Harmonized Scheduling

In our previous discussion, we provided schedulability conditions for basic RHS, but we did not prove any properties on how well the scheme works to save energy. The example of Figure 1 illustrates a situation where basic RHS does, in fact, save energy since all of the idle processor time can indeed be converted into deep-sleep time. However, this does not always occur.

We now extend basic RHS to use a periodic *Energy Saver* task, τ_{sleep}, that is scheduled as the highest priority task with its execution time $C_{sleep} = ST_{sleep}$, a period $T_{sleep} = \mathbf{T_H}$, and phasing $\phi_{sleep} = \phi_1 = 0$. Whenever this task executes, the processor can go into deep-sleep mode

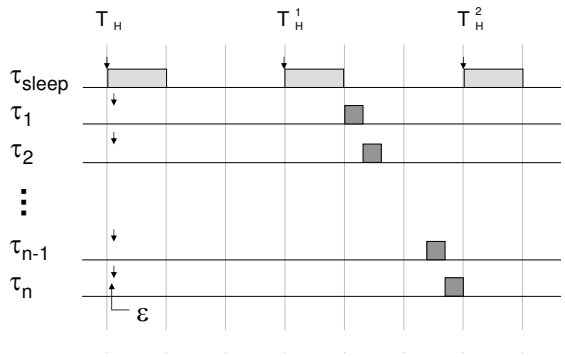

Fig. 2. This figure shows the critical instant when all tasks arrive immediately after $\mathbf{T_H}$.

(for C_{sleep} units of time every T_{sleep} units of time). Real-time scheduling theorists will note that the energy-saver task will indeed behave like a sporadic task [10] executing at the highest priority from a scheduling perspective servicing an endless queue of deep-sleep requests. However, when used with basic rate-harmonized scheduling, the *Energy Saver* exhibits an additional set of very attractive properties. We refer to this hybrid scheme as *Energy-Saving Rate-Harmonized Scheduling*.

D. Terminology

We shall use the following terms.

- The resource being scheduled is said to be *busy* when the resource is executing one or more of the tasks τ_i, $i = 1$ to n. Correspondingly, a *busy duration* is defined to be a contiguous interval in the schedule when the resource is busy.
- The resource being scheduled is said to be in *forced sleeping* mode when the resource is executing τ_{sleep}. Correspondingly, a *forced sleep duration* is defined to be a contiguous interval in the schedule when the resource is in forced sleep.
- A *busy-sleep duration* is defined to be a contiguous interval in the schedule when the resource is either busy or in forced sleep. An *idle duration* is defined to be a contiguous interval when the resource is neither busy nor in forced sleep.

We now present a theorem that allows the coupling of idle durations with forced sleep durations.

Theorem 5. When every job of every task τ_i, $i = 1$ to n, executes for its worst-case execution time C_i, *every* idle duration in the schedule under energy-saving rate-harmonized scheduling will precede (and therefore be contiguous) with a forced sleep execution of τ_{sleep}.

Proof: The Energy-Saver task τ_{sleep} has higher priority than every task τ_i, $i = 1$ to n, and has an initial phasing of $\phi_{sleep} = \phi_1 = 0$. Hence, the resource will be in forced sleep when τ_{sleep} executes at intervals $(kT_{sleep}, kT_{sleep} + C_{sleep})$, $k = 0, 1, 2, \cdots$. Correspondingly, the execution of any job of any task τ_i, $i = 1$ to n, is only during the intervals $[kT_{sleep}+C_{sleep}, (k+1)T_{sleep}]$ for $k = 0, 1, 2, \cdots$.

Consider any time instant t when the resource becomes idle. That is, t represents the beginning of an idle duration. Due to the execution pattern of τ_{sleep}, t must lie within the interval $[kT_{sleep}+C_{sleep}, (k+1)T_{sleep}]$ for some non-negative integer value of k. We will show that the interval $(t, (k+1)T_{sleep}]$ will be an idle duration, which in turn precedes the forced sleep execution of τ_{sleep} during $((k+1)T_{sleep}, (k+1)T_{sleep} + C_{sleep})$.

Since $\mathbf{T_H} = T_{sleep}$, any task τ_i, $i = 1$ to n, that arrives within the interval $[kT_{sleep}, (k+1)T_{sleep}]$ becomes eligible to execute only at $(k+1)T_{sleep}$. So, such an arrival cannot execute in our interval of interest. If task τ_i, $i = 1$ to n, arrived at or before kT_{sleep}, it would have become eligible to execute at kT_{sleep} or earlier. If τ_i has any execution time left at time t, the energy-saving rate-harmonized scheduler must schedule τ_i at time t. This contradicts our assumption that t represents the start of an idle duration. The theorem follows. ■

The worst-case execution time assumption of Theorem 5 can be relaxed, and this is captured in the following corollary.

Corollary 6. Every idle duration in an energy-saving rate-harmonized schedule will precede (and therefore be contiguous) with a forced sleep execution of τ_{sleep}.

Proof: The proof follows the same trajectory as the proof of Theorem 5. Suppose the start-time t of *any* idle duration occurs between kT_{sleep} and $(k+1)T_{sleep}$. All tasks that arrived at or before kT_{sleep} will be eligible to execute and therefore must complete at or before t. All tasks that arrive after kT_{sleep} will be eligible to execute only at $(k+1)T_{sleep}$. Hence, the interval $[t, (k+1)T_{sleep}]$ must be idle, and this precedes the execution of τ_{sleep} in the interval $((k+1)T_{sleep}, (k+1)T_{sleep} + C_{sleep})$. ■

Theorem 7. Every idle duration in an energy-saving rate-harmonized schedule can be used to put the resource into a deep-sleep mode without any time penalty.

Proof: From Corollary 6, *all* idle durations precede (and are contiguous with) a forced sleep execution of τ_{sleep} for a duration of C_{sleep}. This forced-sleep duration of C_{sleep} can be extended to include the preceding contiguous idle duration. This extended deep-sleep duration is longer than C_{sleep}, which guarantees that there is no time penalty switching into and out of deep-sleep mode. ■

Given that *all* idle durations in the energy-saving RHS schedule can be spent in deep sleep, the deep-sleep utilization is given by:

$$U_{sleep} = 1 - \sum_{i=1}^{n} \frac{C_i}{T_i} = 1 - U_{tot}$$

In other words, the deep-sleep utilization is maximal given the taskset utilization of U_{tot}. The only condition that needs to be checked is whether the given taskset is feasible under Energy-Saving RHS.

Theorem 8. A periodic taskset is feasible under Energy-Saving Rate-Harmonized Scheduling if

$$\frac{C_{sleep}}{T_{sleep}} + \frac{C_1}{T_1} \leq 1 \qquad \wedge$$

$$\forall i,\ i = 2 \text{ to } n,\ \frac{C_{sleep}}{T_{sleep}} + (\sum_{j=1}^{i} \frac{C_j}{T_j}) + \frac{T_{sleep}}{T_i} \leq i(2^{1/i} - 1).$$

Proof: Under energy-saving rate-harmonized scheduling, $\phi_1 = \phi_{sleep} = 0$. Also, either $T_1 = T_{sleep}$ or $T_1 = 2T_{sleep}$. Under either of these conditions, τ_{sleep} and τ_1 form a (high-priority) taskset scheduled under rate-monotonic scheduling with harmonic periods. Hence, under RMS theory, if $\frac{C_{sleep}}{T_{sleep}} + \frac{C_1}{T_1} \leq 1$, τ_1 is schedulable. The highest priority tasks τ_{sleep} and τ are harmonic and can be considered to be a single task from the perspective of τ_i's schedulability.

Next, consider an arbitrary task τ_i, $i \neq 1$. Relative to rate-monotonic scheduling, an instance of τ_i encounters a maximum additional delay of $\mathbf{T_H} = T_{sleep}$. Hence, the term T_{sleep} can be added to its computational time of C_i, and RMS utilization bounds can be used for testing feasibility. ■

A less pessimistic schedulability test utilizes the fixed-point approach to determine the exact worst-case response time of task τ_i. To find the worst-case response time of τ_i, let

$$W_0 = C_i + T_{sleep}$$

Iterate on k as follows:

$$W_{k+1} = C_i + T_{sleep} + \lceil \frac{W_k}{T_{sleep}} \rceil C_{sleep} + \sum_{j=1}^{i-1} \lceil \frac{W_k}{T_{sleep}} \rceil C_j$$

until $W_{k+1} = W_k$ in which case the worst-case response time of τ_i is W_{k+1}. Check if $W_{k+1} \leq D_i$. If $W_{k+1} > D_i$, τ_i will miss its deadline and the computation can be stopped.

Remark: A keen reader will note that Theorems 5-8 and the schedulability conditions for energy-saving rate-harmonized scheduling do not require that $\mathbf{T_H} = T_{sleep}$ be $\frac{T_1}{2}$ when $\{\tau_j \mid T_j < 2T_1, j \neq 1\} \neq \emptyset$. In fact, Lemma 3 and Theorem 4 are the only results that require this constraint. Otherwise, if other schedulability conditions are met, one can have $\mathbf{T_H} = T_{sleep} = T_1$ allowing for larger values of C_{sleep}.

Figure 3 illustrates the schedule of a taskset where an increase in the value of C_{sleep} causes the basic RHS scheme to no longer be able to use all idle slots for deep sleep. However, energy-saving rate-harmonized scheduling is able to achieve 100% deep-sleep utilization of all idle slots by delaying yet-to-start tasks into the next T_{sleep} period. The addition of τ_{sleep} into the feasibility conditions does represent a scheduling penalty. However, for many energy-constrained systems like multi-hop sensor networks, the total utilization of the given task set is likely to be 20% or less, and any scheduling penalty only applies when task utilizations are rather high.

E. Energy-Saving RHS with Phase Exploitation

The analysis to date assumes very little about the phasings of tasks except that $\phi_1 = \phi_{sleep} = 0$. When the phasings of other tasks are unknown, worst-case assumptions need to be made. However, the admission criteria can be improved with the initial phasings for all the tasks if the given periodic taskset are known. With this knowledge, the maximum additional delay that a task τ_i encounters relative to RMS scheduling can likely be reduced further below $T_{sleep} = \mathbf{T_H}$. Let the LCM of T_{sleep} and T_i be λ_i. The relative phasings between multiples of T_i and their nearest integral multiples T_{sleep} will repeat every λ_i time-units. The maximal delay between any arrival time of τ_i

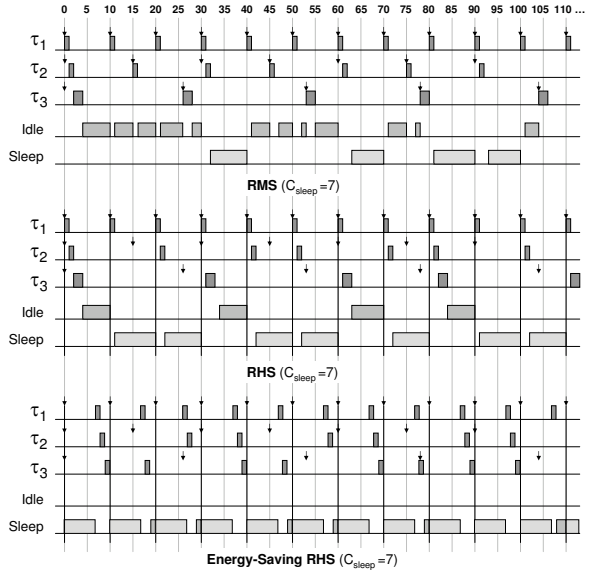

Fig. 3. The taskset $\tau_1 = \{1, 10\}$, $\tau_2 = \{1, 15\}$, $\tau_3 = \{2, 26\}$ with $C_{sleep} = ST_{sleep} = 7$ being scheduled with RMS on the top, basic RHS in the middle and with Energy-Saving RHS on the bottom ($\mathbf{T_H} = 10$). This illustrates an example of when the Energy-Saving RHS schedulability test is satisfied, and RHS is not able to optimally gain sleep cycles. In this example, the exact schedulability test was performed with T_H set to T_1. The exact schedulability conditions work if T_1=T_H always. Note, the total execution time including sleep is not required to be less than T_H. In this case T_3 is preempted after 1 time unit and completes its execution in the next T_H period.

and its eligibility time is given by

$$B_i = max\{(pT_{sleep} - (\phi_i + kT_i)) \mid ((p-1)T_{sleep} < (\phi_i + kT_i)) \wedge (pT_{sleep} \geq (\phi_i + kT_i)), p \in \{0, 1, \cdots, \frac{\lambda_i}{T_{sleep}} - 1\}, k \in \{0, 1, \cdots, \frac{\lambda_i}{T_i} - 1\}\}.$$

This value can be used as the "blocking term" in our feasibility conditions instead of $\mathbf{T_H} = T_{sleep}$. For example, if $\mathbf{T_H} = 6$, $T_i = 15$ and $\phi_i = 3$, $LCM(6, 15) = 30$. The maximum delay encountered by τ_i due to rate-harmonized scheduling is then given by $max((6*1) - (3+15*0), (6*3) - (3+15*1)) = max(3, 0) = 3$, instead of $\mathbf{T_H} = 6$. Hence, the feasibility condition for task τ_i can be improved by a factor of up to $\frac{(6-3)}{15} = 0.2$.

IV. Evaluation

In this section, we compare the performance of rate harmonized scheduling in terms of sleep optimal efficiency and overall impact on the power consumption of a system. We describe the measured energy benefit from a currently deployed sensor networking application as part of the Sensor Andrew project at Carnegie Mellon University.

Sensor Andrew is a multi-disciplinary campus-wide scalable sensor network that is designed to host a wide range of sensing and low-power applications. The goals of Sensor Andrew are to support ubiquitous large-scale monitoring and control of infrastructure in a way that is extensible, easy to use, and provides security while maintaining privacy. Target applications currently being developed include infrastructure monitoring, first-responder support, quality of life for the disabled, water distribution systems monitoring and optimization, building power monitoring and control, social networking, and biometric sensors for campus security. A large component to these applications is an underlying wireless sensor network comprised of the Nano-RK real-time operating system running on the FireFly sensor networking platform.

Nano-RK is a fully preemptive reservation-based real-time operating system (RTOS) with multi-hop networking support for wireless sensor networks. It includes a light-weight embedded resource kernel (RK) with rich functionality and timing support capable of running on low-power micro-controllers. Nano-RK supports fixed-priority preemptive multitasking for ensuring that task deadlines are met, along with support for CPU, network, as well as, sensor and actuator reservations. Tasks can specify their resource demands and the operating system provides timely, guaranteed and controlled access to CPU cycles and network packets. Together these resources form virtual energy reservations that allows the OS to enforce system and task level energy budgets.

In our current Sensor Andrew deployment, the FireFly nodes are battery operated and communicate over multiple hops to a powered gateway that has access to the Internet. The sensor network is primarily designed to efficiently collect sensing data, however it also provides support for various mobile device interactions. We provide a generic communication interface allowing nodes to directly query infrastructure nodes as well as send messages to and from the Internet via the gateway. Communication reservations in Nano-RK provide a mobile node communication budget preventing mobile devices from draining more than their alloted system energy.

The current individual node functionality is supported by the five tasks shown in Table I. The highest priority task consists of a TDMA link layer with a period of 10ms. This period is designed to support each communication slot, however the system can wait multiples of these periods as specified by a communication schedule. The next set

Task Description	C	T	U
Link Layer	3	10	.30
Network Task	1	15	.06
HF Sensor Sampling	1	40	.025
Mobile Node Service	1	300	.003
Diagnostic	1	500	.002

TABLE I

THIS TABLE SHOWS THE WORST-CASE TASK SETS CURRENTLY RUNNING IN NANO-RK AS PART OF THE SENSOR ANDREW PROJECT. DURING TYPICAL EXECUTION MANY OF THESE TASKS ARE OPERATING AT MULTIPLES OF THE MINIMUM PERIODS. ALL EXECUTION TIMES ARE IN MILLISECONDS.

	RMS	RHS	Energy-Saving RHS
U_{sleep}	.67	.98	1.0
Avg. Power (mW)	2.38	2.00	1.98

TABLE II

THIS TABLE SHOWS THE MEASURED VALUES OF THE DIFFERENT SCHEMES GIVEN THE TASK SET IN TABLE I. UNDER THE HIGHEST SYSTEM LOAD WE SEE A 16.8% SAVINGS WITH ENERGY-SAVING RHS AS COMPARED WITH RMS.

of tasks are responsible for managing network routing, sampling high frequency sensor data like the audio sensor and recording run-time diagnostics. The diagnostic task periodically collects information about the system's run-time parameters and eventually writes this to an external flash card. The diagnostic information consists of radio statistics as well as CPU runtime statistics. The radio statistics contain number of transmitted packets, received packets, retries sending a packet due to a dropped ACK and packet loss as well as average signal strength values between neighbors. The CPU statistics keep track of each tasks utilization as well as the time the processor spends in idle as compared to deep sleep. Using these values, we calculate that gain of using Energy-Saving RHS as shown in Table II. Under normal operation, many of these tasks are not operating at every period. For example, the *High Frequency (HF) Sensor Sampling Task* would normally only execute at this frequency when audio data is requested from the gateway. Given the worst-case situation when all of these tasks are executing (which does occur occasionally), Energy-Efficient RHS saves up to 16% as compared to RMS.

Our experimental numbers are based on the ATmega1281 processor, however Table IV shows the corresponding parameters for other processors. In all of these cases, disabling the oscillator to enter the deepest sleep mode consumes proportionally less energy than the processor's idle mode. Even in cases where the idle energy of a processor is quite low, RHS can be used to further improve energy performance with little overhead.

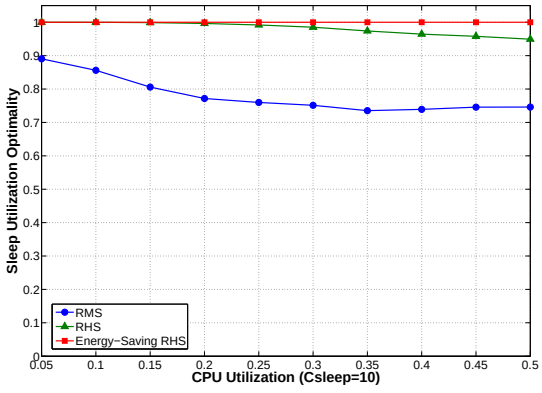

Fig. 4. This figure shows how close to optimal each scheme performs with respect to converting idle processor time into sleep time with a C_{sleep} value fixed at 10. Each point in the graph represents the average of 1000 simulated schedules.

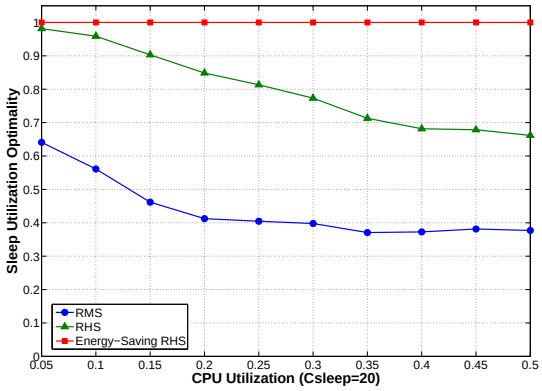

Fig. 5. This figure shows how close to optimal each scheme performs with respect to converting idle processor time into sleep time with a C_{sleep} value fixed at 20.

Task	C	T	U
1	1	20	.050
2	1	25	.040
3	1	26	.038
4	1	28	.035
5	1	32	.031
6	2	50	.04
7	2	67	.029
8	3	91	.033
9	9	100	.090

TABLE III

THIS TABLE SHOWS A TASK SET THAT WHEN $C_{sleep} = 10$ AND $T_{sleep} = 20$, THE POWER CONSUMPTION FOR A FIREFLY NODE WOULD BE 9.83 mW FOR RMS AND 6.5 mW FOR ENERGY-SAVING RHS. IN THIS EXAMPLE, ENERGY-SAVING RHS CONSUMES 33% LESS TOTAL ENERGY.

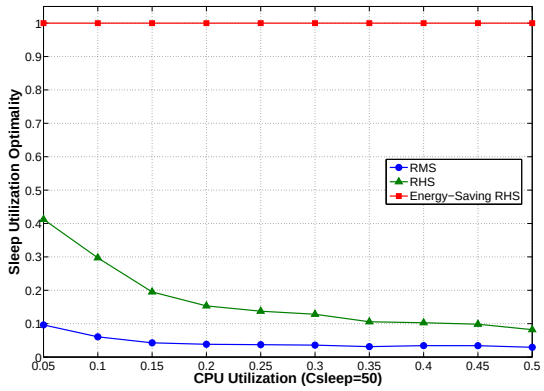

Fig. 6. This figure shows how close to optimal each scheme performs with respect to converting idle processor time into sleep time with a C_{sleep} value fixed at 50.

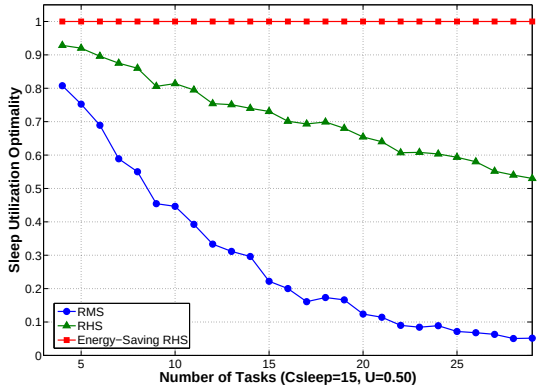

Fig. 7. This figure shows the effect of increasing the number of tasks given a fixed C_{sleep} value of 15 and a fixed CPU utilization of 50%.

Processor	Freq. (MHz)	Power Sleep (uW)	Power Idle (mW)	Power Active (mW)	Sleep to Idle (ms)	Idle to Active (us)
ATmega1281	8	16	6.6	23	12	6
Hitachi H8	8	.05	60	90	100	8
MSP430F5418	8	.33	.0085	4	10	5
ST Cortex M3	20	5.6	18.5	85	2	2
LPC2106	60	1	10	108	10	4
BF531	600	15	30	616	5	2

TABLE IV
THIS TABLE SHOWS THE ENERGY AND STATE TRANSITION TIMES OF VARIOUS MICROCONTROLLERS. NOTE, MANY PROCESSORS HAVE MULTIPLE OPERATING FREQUENCIES. THIS TABLE SHOWS AN ESTIMATE OF A SINGLE SAMPLE OPERATING POINT.

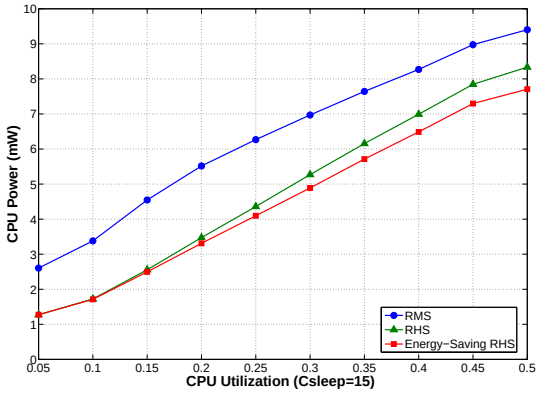

Fig. 8. This figure shows the CPU power consumption given the FireFly hardware parameters under the different schemes. Each point in this graph is the average of 1000 randomly generated task sets with n between 5 and 15, period between 1 and 200 with a C_{sleep} fixed at 15ms.

A. RHS Performance

In this section we discuss various trends apparent in Harmonized Scheduling of tasks. We performed a set of experiments based on a large number of uniformly distributed random task sets. Each task set was given a period between 1 and 200 time units and simulated over its entire hyper-period. We define the metric *Sleep Utilization Optimality* which is the ratio of the total actual deep sleep time to the total non-busy time in a hyper-period. A value of 1.0 indicates that all available idle time was used for deep sleep.

In our first set of experiments, we adjust the C_{sleep} parameter and look at how CPU utilization affects the Sleep Utilization Optimality of the processor. During these tests, the number of tasks was randomly selected to be between 5 and 15. We see that smaller C_{sleep} values tend to work reasonably well with RMS because small intervals between tasks can be converted to sleep time. If C_{sleep} is set to 1, then all schemes would perform identically. As shown in Figure 5 and Figure 6, as the C_{sleep} value increases to 20 and 50 the effectiveness of using RHS becomes more prominent. It is also clear that as the workload increases, the gain improves.

Figure 7 shows how the number of tasks executing in the system also has an impact on the ability for the system to sleep. In this experiment, the C_{sleep} value was fixed at 15 (the real value for our system) and the workload was fixed at 50% utilization. As the number of tasks increase, RMS begins to rapidly deteriorate in performance while Energy-

Saving RHS maintains its optimal sleep utilization. As the number of tasks increases, the number of preemptions and likely phase offsets also increases. The difference between the schemes as the number of tasks in the system inceases shows that Energy-Saving RHS is highly scalalble.

Figure 8 shows the overall impact of the schemes with respect to CPU power consumption. This figure characterizes the design parameters from the FireFly v2.2 hardware. The active energy of the processor is $19.8mW$, the idle energy is $6.6mW$ and the sleep energy is $6.6\mu W$. The C_{sleep} parameter is set to $15ms$. We see that with random task sets, even at a low-load that the savings can be quite significant, up to 39%. Table III shows an example where Energy-Saving RHS saves 33% of the total power RMS would consume. As the workload increases, we do not see as much divergence of the lines as in the previous examples due to the dominating active power term as the load increases. As the active energy in the system approaches the idle energy, then RHS schemes will perform even better. This will be a natural trend in micro-controllers since the silicon process technology is improving making the clock crystal a dominating factor in power consumption. Since sleep modes typically disable the oscillator, the sleeping mode should remain significantly less than active and idle.

V. Conclusion

In this paper we introduce a novel but simple, practical and effective technique that maximizes the percentage of time a processor spends in deep sleep without violating the timing constraints of the given real-time task set. We introduce a class of Rate-Harmonized Schedulers that adjust phasing between periodic tasks in order to remove idle slots between active execution that are too short for the processor to make a round-trip transition from idle into deep-sleep. One particular instance of these schedulers, *Energy-Efficient RHS* , provably has the property that *every* idle time-unit not spent in active processing can be converted directly into deep-sleep time for the processor. Not only does this improve energy-efficiency, but it simplifies scheduler design and can even potentially eliminate the idle state of the processor to save on hardware complexity. We provide theoretical analysis and experimental evaluation of these schemes as applied to sensor networks where energy is highly important.

References

[1] A. Eswaran, A. Rowe and R. Rajkumar. Nano-RK: an Energy-aware Resource-centric RTOS for Sensor Networks. *IEEE Real-Time Systems Symposium*, 2005.

[2] Rajkumar R. Oikawa S. Portable RK: A Portable Resource Kernel for Guaranteed and Enforced Timing Behavior. *IEEE Real-Time Technology and Applications Symposium (RTAS)*, 1999.

[3] http://www.nanork.org/nano-RK/wiki/FireFly (viewed 5/20/2008).

[4] Woo A. Hollar S. Culler D. Pister K. Hill J., Szewczyk R. System Architecture Directions for Network Senors. *ASPLOS*, November.

[5] "http://focus.ti.com/docs/toolsw/folders/print/z-stack.html (viewed 5/20/2008)".

[6] Carlson J. Dai H. Rose J. Sheth A. Shucker B. Deng J. Han R. Abrach H., Bhatti S. MANTIS: System Support For MultimodAl NeTworks of In-situ Sensors. *2nd ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, 2003.

[7] Shea R. Kohler E. Srivastava M. Han S., Rengaswamy R. SOS : A Dynamic Operating System for Sensor Nodes. *In Third International Conference on Mobile Systems, Applications and Services (Mobisys)*, 2005.

[8] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *Journal of the ACM, V20, N1*, pages 46–61, 1973.

[9] Lehoczky, J. Fixed priority scheduling of tasks with arbitrary deadlines. *Proceedings of the Real-Time Systems Symposium (RTSS)*, 1990.

[10] Sprunt B., Sha L., Lehoczky J. Scheduling Sporadic and Aperiodic Events in a Hard Real-Time System . *CMU/SEI-89-TR-11 Carnegie Mellon University*, 1989.

[11] B. M. Gordon R. Gonzalez and M. A. Horowitz. Supply and threshold voltage scaling for low-power CMOS. *IEEE Journal of Solid-Sate Circuits, vol. 32(8)*, 1997.

[12] Rajkumar R. Saewong S. Practical Voltage-Scaling for Fixed-Priority RT-Systems. *In Proceedings of the 9th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2003.

[13] Joseph M., Pandya P. Finding Response Times in a Real-Time System. *The Computer Journal*, 1986.

[14] Tindell, K. W. An Extensible Approach for Analysing Fixed Priority Hard Real-Time Tasks. *Department of Computer Science, University of York, UK.*, 1992.