Power and Control in Networked Sensors

E. Jason Riedy
ejr@cs.berkeley.edu

Robert Szewczyk
szewczyk@cs.berkeley.edu

11 May, 2000

Abstract

The fundamental constraint on a networked sensor is its energy consumption, since it may be either impossible or not feasible to replace its energy source. We analyze the power dissipation implications of implementing the network sensor with either a central processor switching between I/O devices or a family of processors, each dedicated to a single device. We present the energy measurements of the current generations of networked sensors, and develop an abstract description of tradeoffs between both designs.

1 Introduction

Over the last few decades, "Moore's Law" enabled the hardware engineers to put a substantial amount of computation and storage into increasingly smaller packages. Additionally, advances in CMOS processing and MEMS research make it possible to construct a low-cost networked sensor. In the near future, researchers predict that it will be possible to integrate communication, power sources, sensors and actuators with computational elements in a mm^3 [11].

Energy stored within each networked sensor is the most precious resource, so both the hardware architecture and the software system should be optimized for its usage. Each sensor has a limited energy source, and replenishing this energy source may be either impossible (no physical access to the device) or not economically viable (the maintenance cost can exceed the sensor cost by orders of magnitude). Thus the energy efficiency is probably the most important metric against which the architectural choices must be evaluated.

A typical desktop PC contains many different processing elements: besides the CPU, there are many dedicated I/O processors for handling graphics, network traffic, or hard disk requests. These dedicated I/O processors were added in order to enhance the performance of the system. In contrast, the current generation of the networked sensor system looks more like primitive home computer system from the late '70s: there is a single processor handling all the I/O devices. The architects of the networked sensor will sooner or later face a design dilemma: should there be a dedicated processing element for each I/O device or should the management of the I/O devices be centralized? How should these decisions be evaluated? What are the fundamental tradeoffs between these design alternatives?

In this paper we compare and analyze two architectures for networked sensors: one based around a single CPU handling multiple I/O devices, and one based around two general purpose processors: one handling the wireless communication system, and one handling other I/O devices.

In general the power dissipation within a system is proportional to frequency. In a system with real-time deadlines, assigning the tasks to dedicated processors implies that the individual processors will be able to run at a significantly lower frequency. Lowering the frequency can lead to lowering the operating voltage of the component. These two factors could yield substantial power savings. Furthermore, since the individual components might be tuned to meet their deadlines just in time, they would not waste any energy in the idle states, whereas the scheduling of tasks on a single processor might require that this processor spends a portion of time idle.

These potential savings need to be balanced against several factors: the communication between the processors will almost certainly not be free, there typically is a fixed cost associated with having an extra component. Allocating tasks to processors is quite similar to a packing problem. Depending on the granularity of tasks and a particular split, multiple processors allocated to the problem might have to run at either higher or lower cumulative frequency than a single processor allocated to the task.

In this paper, we examine the implications of both the single and multiple processor architectures on the power dissipation within the system. We analyze both design styles in terms of abstract models, and compare these models with the measurements of a real system: a prototype networked sensor [9] running TinyOS [6]. The rest of this paper is organized as follows. Section 2 introduces simple models of hardware and software of the networked sensor and Section 3 presents a particular implementation of a networked sensor used to ground our empirical study. Section 4 analyzes the power consumption of a single processor system: we present a set of detailed energy measurements of the single processor design in Sections 4.1 and 4.2, and analyze this data in abstract terms in Sections 4.3 and 4.4. Section 5 extends this analysis to systems with multiple processors. In Section 6 we describe the related work, and conclude in Section 7.

2 Architecture Model

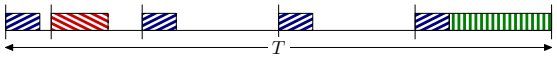
To draw sound general conclusion about the power analysis in networked sensors we need to be able to abstract the observations of our particular hardware and software system. In this section we present the models of both hardware and software.

2.1 Task Model

A light-weight networked sensor is not expected to be a general purpose computing device. Its goals are to collect readings, process them slightly, and communicate readings with other sensors. For example, a node may take three temperature readings, average them, and trade averages with its neighbors. These goals can be subdivided into various tasks, and the tasks recur periodically and often be subject to real-time constraints. Figure 1 shows a slice of a sensor's activities over a time span T. For analysis, we assume that this time span is completely periodic; the same tasks are executed in the same number for each slice of length T.

Over the time T, the processor executes instructions to control the devices and process readings. Let K be the number of clock cycles occupied by instructions during T. At a particular execution frequency of f cycles per second, let ρ be the utilization,

$$\rho = \frac{K}{fT}. \tag{1}$$



Sensing ( ), communicating ( ), and processing ( ).

Figure 1: A networked sensor runs only a few classes of tasks.

Note that K/T is a frequency itself, and at that frequency the K cycles span the entire time T. This frequency, f_m, is the minimum frequency supporting the given task set, and $f\rho = f_m$. Relating the frequency and utilization will be useful for expressing energy usage. When the processor is not busy, $1 - \rho$ of the time, it sits in a low-power idle or stop mode.

Scheduling general tasks to meet real-time constraints is a challenging topic in its own right, one we leave to others [14, 3]. We assume that a scheduling exists for any particular processor and device configuration. This is not entirely realistic, but it allows us to focus on power consumption rather than real-time scheduling.

2.2 Hardware Model

To examine the power consumption in a small, networked sensor, we also need a simple model of the hardware. Figure 2 shows the block structure of sensor nodes with 'dumb' and 'intelligent' I/O devices. The devices are assumed to have power needs that do not vary as instructions are partitioned between processors. This is reasonable if the devices' activity is managed entirely with respect to real time, as when a temperature sensor is be run every half-second for a tenth of a second. The energy consumed by operating the sensor does not depend on the number of processors or partitioning of tasks, and so we do not complicate our analysis with these constants. We also assume that the switching frequency of the processor pins connected to the devices is determined by the devices and is also constant for a task set.

Now how much energy is needed for the processing? The energy consumed is the power over the time period, $E = \int_0^T P dt$. For the DC processing components, $P = IV$, where the current I is a function of f. We assume that voltage and frequency vary indepently, *e.g.* we remain away from fundamental CMOS limits. We hold the voltage V constant, so $E = V \int_0^T I(f) dt$.

The task model, Section 2.1, separates the current

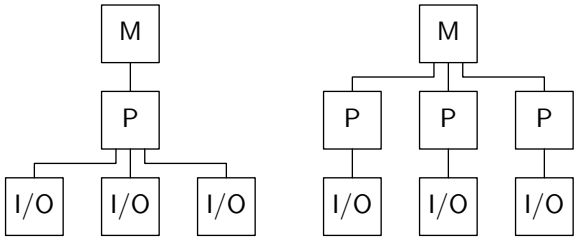


Figure 2: One central processor controls many 'dumb' devices, while each 'intelligent' device has a dedicated processor.

into an active current for ρT and an idle current for $(1-\rho)T$. This gives an energy consumption of

$$\begin{aligned} E &= VT(\rho I_{\text{active}}(f) + (1-\rho)I_{\text{idle}}(f)) \\ &= VT(I_{\text{idle}}(f) + \rho(I_{\text{active}}(f) - I_{\text{idle}}(f))) \\ &= VT(I_I(f) + \rho I_A(f)) \end{aligned} \quad (2)$$

over the recurrent period of length T. Here we're defined I_I as the idle current, and I_A as the extra current needed over I_I for executing instructions. With $I_A = I_{\text{active}} - I_{\text{idle}}$, I_A expresses the processor's activation current.

In Equation 2, the voltage and time are held constant, so the only quantity left to optimize is the drawn current. For the processors of interest, I is a *linear* function of f, and $I_*(f) = a_* f + b_*$, where $*$ is either I or A for the idle and activation currents, respectively. Later sections will investigate the current for a given task set, looking for optimal parameters and comparing different processor configurations. Note that the derived power dissipations are for the power lost through the processor, not necessarily the power drawn from the battery. Others have examined that problem[10], and minimizing the current also maximized the battery life in the proposed models.

3 Experimental Platform

Having introduced the abstract model, let's examine how well they map onto the real system. Below, we examine a simple application for a networked sensor developed in [6]: the application measures environmental parameters periodically, broadcasts these measurements over the low power RF link, participates in routing protocols, and responds to data queries. This high level specification was implemented on a SmartDust prototype [9] using the TinyOS framework [6]. In this section we briefly present the hardware and software environments.

3.1 Hardware organization

Our sample networked sensor consists of a microcontroller with internal flash program memory, data SRAM and data EEPROM, connected to a set of actuator and sensor devices, including LEDs, a low-power radio transceiver, an analog photo-sensor, a digital temperature sensor, a serial port, and a small coprocessor unit. While not a breakthrough in its own right, this prototype forces us to reason about the various parts of the system.

The single most important component of the system is the radio. It represents an asynchronous input/output device with hard real time constraints. It consists of an RF Monolithics 916.50 MHz transceiver (TR1000) [12], antenna, and collection of discrete components to configure the physical layer characteristics such as signal strength and sensitivity. It operates in an ON-OFF key mode at speeds up to 19.2 Kbps. Control signals configure the radio to operate in either transmit, receive, or power-off mode. The radio contains no buffering so each bit must be serviced by the controller on time. Additionally, the transmitted value is not latched by the radio, so jitter at the radio input is propagated into the transmission signal.

The processor is an Atmel AVR 90LS8535 [2] externally clocked at 4MHz. It is an 8-bit Harvard architecture with 8KB instruction and 512 bytes of data memory. The processor integrates various kinds of peripherals: a UART controller, an A/D converter, several timers, and general I/O pins. Noteworthy are the sleep modes supported by the processor: *idle* shuts down just the processor, *power down* which shuts off everything but the watchdog and asynchronous interrupt logic necessary for wake up, and *power save*, which is similar to the power down mode, but leaves an asynchronous timer running. The latter two modes reduce the energy dissipation by a factor of a 1000, but, unfortunately, it takes miliseconds to restore the processor from the deeper sleep modes. Also significant is the current drawn by each of the sleep modes as a function of processor frequency: in the idle mode, the current is linearly dependent on frequency of the processor, whereas in the latter two modes the current is independent from frequency. This distinction becomes significant below, in Section 4.3.

The temperature sensor (Analog Devices AD7418)

represents a large class of digital sensors which have internal A/D converters and interface over a standard chip-to-chip protocol. In this case, the synchronous, two-wire I^2C [13] protocol is used with software on the microcontroller synthesizing the I^2C master over general I/O pins.

The light sensor is a photoresistor with resistance ranging from 10Ω to 50kΩ. It forms a voltage divider with a fixed resistor, and an A/D converter inside the microcontroller is used to read the light levels.

3.2 TinyOS

As a model of execution for the network sensor application, we chose the TinyOS described in [6]. TinyOS is a small operating system designed with several goals in mind: providing support for highly concurrent applications, providing system modularity with minimal overhead, and placing minimal requirements on the underlying hardware, both in terms of the program size and in terms of other computational resources. The execution model provided by TinyOS is similar to FSM models, but considearbly more programmable.

A complete TinyOS system consists of a simple scheduler, and a graph of *components*. Each component has four interrelated parts:

1. an encapsulated fixed size *frame*. The use of static memory allocation allows for checking the memory requirements, and elimination of overhead associated with dynamic memory allocation

2. a set of *event handlers*, which are typically invoked in response to hardware events. Typically the responsibility of the thread is to deposit information within the frame, schedule threads for more complex processing of data, and signal higher-level events.

3. a set of *commands*, which are non-blocking requests to lower level components. Commands can schedule threads and call other commands,but they may not signal events.

4. a bundle of simple *threads*, which are primarily responsible for the computation within the system. Threads run to completion, but they can be preempted by event handlers. Run-to-completion semantics allows for maintaining a single stack, which is important in a memory constrained system. Threads allow for simulating concurrency within each component, since they execute asynchrounously with respect to events. Threads should never spins or wait for a condition, instead the scheduling mechanism should be used.

The thread scheduler is currently a simple FIFO scheduler, utilizing a bounded size scheduling data structure. Depending on the requirements of the application, more sophisticated priority-based or deadline-based structures can be used. Within the current TinyOS version, the scheduler puts the processor into an idle mode.

Currently, the components available within TinyOS can be divided into three groups. First, components that are a thin abstraction over hardware; the UART interface, a simple I/O pin or a timer fall into that category. The components in the second group act as a replacement for unavailable hardware, for example the byte-level radio controller implements the functionality similar to that of a UART on top of a bit level radio component. Another component falling into this category is the I^2C, which implements that protocol in software.

Finally, the third group consists of high level software components. These components perform routing, control and data transformations. The active message component serves as an example for this group, since it provides dispatch and routing.

Each component describes both the resources it provides and the resources it requires. This makes it quite easy to wire the components together, and enables the use of higher level design tools. Communication between components takes the form of a function call, which provides compile-time type checking and has low overhead.

3.3 Application implementation

The application running on the networked sensor monitors the temperature and light conditions and periodically broadcast their measurements onto the radio network. Furthermore, each sensor is configured with routing information that will guide packets to a central base station. Thus, each sensor can act as a router for packets traveling from sensors that are out of range of the base station.

There are three I/O devices that this application must service: the network, the light sensor, and the temperature sensor. Of these, the network is the most

complex. As pointed out above, the RFM radio only provides a bit level interface, which imposes strict real time limits on the application. In order to provide the communication, the radio input is sampled by software at the rate of 10000 times per second. The bits are decoded [1] and assembled into bytes. On a higher level, the bytes are assembled into packets, and dispatched depending on their type and destination. All layers from sampling bit-level input to dispatching packets are performed in software. This function maps very cleanly onto the abstract task model described in Section 2.1: the amount of work is very periodic, since we need to perform a fixed amount of work per bit, a fixed amount of work per byte and so on. The TinyOS events map quite well onto the periods of sensing, while the TinyOS threads map onto computation blocks in the abstract model. The real-time constraints are quite severe: the handler sending and receiving bits does have enough time to receive and store the bits, but it cannot perform the signal decoding without missing a deadline. In order to cope with this problem, the encoding and decoding of bits are done within a thread rather than within the event handler (see Section 4.4 for the power implications of this constraint).

The temperature sensor uses the I^2C protocol to communicate with the rest of the system. While this protocol is perhaps more complex than the simple encoding used by the radio, it has a flexible timing model, which implies that the real-time deadlines will be much more forgiving.

The light sensor is the simplest of the I/O devices: currently it is connected to the A/D converter in free run mode. Reading data involves simply reading an appropriate register of the A/D converter.

Since the components of TinyOS are well isolated from one another, it is an easy task to partition the task between multiple processing units. Furthermore, the asynchronous nature of the entire system ensures that the natural partitioning along the component boundaries is quite efficient.

4 One Processor

To start our study of power usage on a single processor system, we present the measurements of the experimental system. To gain an understanding of the relative energies spent on accessing the sensors, and processing the incoming data, we developed a set of microbenchmarks measuring various primitive operations. As these benchmarks show, the processor is in fact consuming a significant amount of energy compared with the I/O devices, thus the question of optimizing the processor usage is a valid one. We then proceed to the measurements of the TinyOS application: we analyze the overheads of the system and extract the real-time tasks fitting within the model presented in Section 2.1. We then proceed to abstract analysis of the one processor problem.

4.1 Microbenchmarks

The processor on the mote is externally clocked at 4MHz, and connected to a 2.84V power supply. We placed a 10 Ω resistor in series with the mote (placed between ground and the device), and measured the voltage drop across the resistor to arrive at the current drawn by all modules on the device. The measurements were made with the aid of an HP 16550A logic analyzer / 16532A digital oscilloscope, which was triggered by the benchmarks by one of the otherwise unused pins on the processor. The oscilloscope outputs a picture of voltage samples. This picture is downloaded to a PC, the points converted to Amps (as we know that the voltage drop is measured across a 10 Ω resistor) and integrated between two triggers points.

All of our benchmarks work by taking the difference between an execution which includes either a known number of the instruction of interest or a known operation on a module and an execution without it. They all have the following form:

```
Turn_off_all_devices
Setup
While(1) {
 Flash_Trigger_Pin
 Body
}
```

For testing specific instructions (`InstX`), `Setup` is blank while `Body` has the general form of:

```
For i=0 to N {
 InstX
 InstX
 ...
 InstX
}
```

[1] The radio requires a DC balanced signal. Currently TinyOS supports Manchester and 4B6B encodings.

Instruction type	Energy per cycle (nJ)	Energy per instr (nJ)
idle	1.70	1.70
noop	3.39	3.39
arithmetic/logic	3.41	3.41
memory read*	3.66	7.32
memory write*	3.75	7.50
Device	**Energy per CPU cycle**	**Energy per quantum**
LED	1.89	1.89 nJ/cycle
Photo	0.08 - 0.28	0.08 - 0.28 nJ/cycle
ADC	0.36 - 0.30	4.62 - 3.95 nJ/conv.
RFM send	2.56	2050 nJ/bit
RFM receive	2.44	1950 nJ/bit

Table 1: Energy consumption of instructions (left) and external modules (right). RFM send and receive measurements were done assuming 100 μs pulses.

`InstX` is executed multiple times per iteration to make it a more significant portion of the total computation (since some cycles go to the loop overhead). This is then compared with running the empty loop.

Although the benchmarks for the modules vary slightly depending on the module, the light sensor serves as a good example. This module requires 5 measurements:

1. Base: Setup is blank - nothing is turned on
2. Light Sensor without ADC in the dark: Setup turns on the light sensor but not the ADC to convert the output, and the sensor is covered
3. Light Sensor without ADC in full light: Same as above except a bright light is shown on the sensor
4. Light Sensor with ADC in the dark: Same as 2 but the ADC is activated to convert the analog signal
5. Light Sensor with ADC in full light: Same as 3 but with the ADC

Since the light sensor is a photoresistor, the current drawn depends on the light level. The last two measurments are needed because the amount of current drawn by the ADC is also effected by the light level (see below). The body in each of these tests is a busy loop of a fixed length. To compute the energy consumed by each component, we take the difference of the energy used when that component was active at a certain light level and when it was not integrated over the same period.

Table 1 summarizes our findings from running these microbenchmarks. We found that arithmetic/logic operations consumed about the same amount of energy as noops, while loads and stores were only slightly more expensive per cycle. Note, however, that loads and stores take two cycles. This was rather surprising at first, since we expected that going to memory would be much more expensive than accessing registers. However, a closer look at the design of the ATMEL AVR architecture reveals that the register file, the I/O registers and the data memory are located in a single block of on-chip SRAM. The slight overhead seems to be caused by the transfer of the data words through the main data bus, rather than through a dedicated bus used to read the register file.

Another surprise was that communication (either directly through a pin, or through a serial interface such as the UART) did not consume any additional energy (note that our measurement uncertainty is around 0.05nJ/cycle). This is significant for our analysis of multiple processor architectures.

4.2 TinyOS measurements

In our previous work [6] we presented basic measurements of the TinyOS system. The previous measurements were taken using a logic analyzer. In the context of this work, we enhanced ATMEL AVR simulator [4] with a set of I/O devices, and used the instruction traces to analyze the performance of TinyOS components. In general, we find an excellent agreement between the data gathered in this experiment and the data gathered before. Table 2 shows the costs of some fundamental operations on the ATMEL AVR architecture. While small in absolute terms, these numbers can add up to a significant percentage of the total execution time of an application. For example, consider the data presented in Table 3. On average, it takes 163 cycles to send a bit: 60 cycles are spent saving and restoring state and the control flows through several modules (which implies not only posting of several commands but also state transitions within several distinct state machines). All this effort is required to set an output pin depending on a value somewhere in memory, and to update some state variables, all of which tasks should never take more than 50 cycles. This disparity shows the importance of the structuring of the application and the importance of inter-component optimization. We discuss the impact of these overheads below, in Section 4.4.

Based on these figures, it appears that TinyOS actively looking for a start symbol within incoming

Operation	Cost (cycles)
Byte copy	8
Post an event	10
Post a command	10
Post a thread to scheduler	46
Context switch overhead	51
Interrupt (hardware overhead)	9
Interrupt (software overhead)	60

Table 2: Cost of the basic operations and overheads within TinyOS. Note in particular the relatively high cost of the software overhead within interrupt handlers. This overhead is caused by saving the processor state; it becomes quite significant in many of the handlers actually executed within TinyOS

Task	Avg. Time (cycles)	Max. Time (cycles)	Period (μs)
Start receive	130	144	50
Receive bit	191	315	100
Send bit	163	301	100
Byte encode	130	130	1600
Byte decode	146	146	1600

Table 3: Real-time tasks within TinyOS. Byte encode and decode are structured as TinyOS threads, their timing does not include the time scheduling and switching time. On the other hand, the other operations are events, which are executed within the context of an interrupt. Their timing information does include the software overhead of saving the state.

transmission pushes the limits of the hardware. If TinyOS were purely event based, then the processor running at 4 MHz is used about 75% of the time. However, the situation is a bit more complex, since TinyOS supports tasks, and contains a scheduler. Execution of an empty scheduler loop takes 40 cycles; that number should be added to the average execution time in order to compute the average time spent working. In this case, we can sleep only for 15% of the time.

The large disparity between the maximum and average times of send and receive bit operations are caused by the fact that when the bit-level processing layer completes a byte, that triggers an event propagation through many layers. Separating the network stack between multiple processors helps to reduce the disparity between the average and maximum times.

4.3 Simple Analysis of One Processor

In a real-time application on a single processor the main parameter we can adjust is the frequency of the processor. The frequency impacts several parameters within the system: the time spent in active and idle modes, and the amount of overhead inherent in a particular structure of the application. In the next two sections we present the analysis of how frequency and system overhead impact the power usage.

We begin with the current consumption of a single processor running a fixed set of tasks. This simple situation serves as base for later comparisons. The task set will have a constant number of active cycles K over time T, and f_m is a constant. Expanding the idle and activation current gives the total current consumption as a function of the utilization ρ,

$$\begin{aligned} I &= (a_I f + b_I) + \rho(a_A f + b_A) \\ &= (a_I f_m/\rho + b_I) + \rho(a_A f_m/\rho + b_A) \\ &= b_I + a_A f_m + a_I f_m \rho^{-1} + b_A \rho. \end{aligned} \tag{3}$$

Of $a_I \neq 0$, as in the Atmel's idle mode, then the current follows the solid curve in Figure 3. The curve has a minimum at

$$\rho_{\text{opt}}^2 = \frac{a_I}{b_A} f_m. \tag{4}$$

Substituting this back into the current yields the optimum current for the task set, $I_{\text{opt}} = b_I + a_A f_m + 2\sqrt{a_I b_A f_m}$. Because $\rho \in (0, 1]$, the optimum value can only be achieved when

$$f_m \leq \frac{b_A}{a_I}. \tag{5}$$

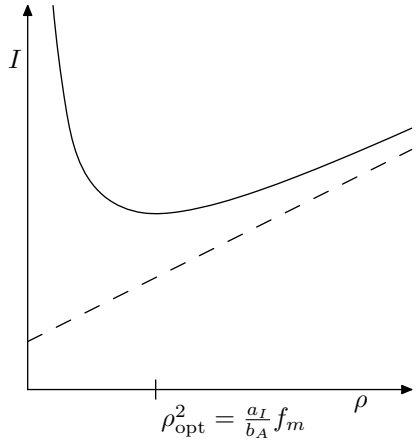


Figure 3: The total current follows the solid line when the idle current depends on frequency (idle mode) and the dashed line otherwise (stop mode).

	Optimal Utilization (%)	Optimal Frequency (MHz)	Energy Usage (mJ)
Base TinyOS	90.6	2.22	0.42
cheap interrupts	76.5	1.87	0.34
cheap scheduling	67.6	1.65	0.30

Table 4: Optimal utilization and energy usage for sending data within TinyOS with varying overhead structures.

Sensing (▨), communicating (▨), processing (▨), and **overhead** (■).

Figure 4: Events incurr some overhead processing for every context switch.

When $f_m > b_A/a_I$, the least current draw occurs with the processor running as slowly as possible. In the extreme case, $\rho = 1$, and $I_{\text{opt}} = b_I + b_A + (a_I + a_A)f_m$. Real-time constraints will force $\rho < 1$ in many circumstances.

However, if the processor is put into a stop mode when inactive, one that draws a current invariant with the active frequency, then $a_I = 0$. The current follows the dashed line in Figure 3. The minimum would be at $\rho_{\text{opt}} = 0$ and $I_{\text{opt}} = b_I + a_A f_m$. Zero utilization requires an infinite frequency and is not achievable. The least current draw here occurs when running the processor as fast as possible. Frequency becomes free, although it will be limited by the operational voltage at the very least.

To make this analysis more concrete, Table 4 shows the optimal utilization and the expected energy usage for our experimental sensor. The table presents the data ignoring some scheduling contstaints: in particular, since maximum execution time is larger than the average, running at the optimal frequency might cause us to miss deadlines. Restructuring the application across multiple processors might help with this disparity by allowing each of the processors to run closer to the optimal frequency while still meeting deadlines.

4.4 One Processor with Overhead

The previous analysis ignored the costs of context switching and scheduling. Each event swaps processor state into memory on entry and out of memory on exit. The time and instruction overhead can be modeled by splitting the required cycles into work and overhead cycles, $K = K_w + K_o$. The processor utilization then splits into $\rho = \rho_w + \rho_o$. Some of the work cycles may be considered as 'overhead' from different viewpoints, but we strictly limit the definition of overhead to those cycles that are an artifact of a particular execution environment. To us, the cycles spent saving registers to memory during a context switch are overhead, but instructions that update a loop variable in some processing thread are not.

The current consumption with overhead is modeled by

$$I = I_I + I_A(\rho_w + \rho_o). \tag{6}$$

Some overhead is avoidable. Imagine running the processor in Figure 4 just a little faster, fast enough that the processing after the last sample could complete before the next sample. Then all events could avoid context switches. Limit K_o to only the cycles spent swapping processor state. An increased frequency cf, $c > 1$, can trade the overhead instructions for some extra idle time at the higher frequency.

If it is possible, when is it beneficial? Let

$$I'(cf) = I_I(cf) + I_A(cf)\rho_w, \tag{7}$$

eliminating all of the overhead at cf. Then the question can be rephrased as finding the conditions when

$$I'(cf) < I(f). \tag{8}$$

Intuitively, this should occur when the extra current needed for higher-frequency work is less than the cur-

rent needed for the overhead. Indeed, when we expand Equation 8, we find that the condition becomes

$$(c-1)(a_I + a_A\rho_w)f < I_A(f)\rho_o. \quad (9)$$

The right side, $I_A(f)\rho_o$, is the overhead current, and the left, $(c-1)(a_I+a_A\rho_w)f$, is the extra work current at the higher frequency. So when the condition in Equation 9 is satisfied, running at a higher frequency cf can *decrease* power consumption by $I_A(f)\rho_o-(c-1)(a_I + a_A\rho_w)f$.

5 Multiple Processors

The decision to use multiple processors is normally driven by the need to meet real-time scheduling constraints at a lower processing frequency. How does it affect the power consumption?

5.1 Multiple processor experiment

To experiment with a multiple processor design we decided to simulate to simulate the processors on a simulator. Rather than looking at the dynamic interaction between the dedicated processors, we analyzed the traces generated by a single processor running augmented pieces of TinyOS application. When faced with a multiple processor problem, the designer must ask fundamental questions: How is the application partitioned? What are the interprocessor communication costs? The design we experimented with is only a single point, the more general implications of multiprocessor designs are explored in the Sections below.

In order to partition the TinyOS application we have developed a light-weight RPC-like component. At the moment it is only capable of transmitting scalar parameters. In our experiments, this limited capability still allowed for a substantantial freedom in choosing the partitioning of components. Our measurements from Section 4.1 show that the communication between two processor over an external bus is essentially free in hardware sense (at least at low data rates). Similarly, our measurements of memory operations on the ATMEL show that accessing memory (and by extension, communication through shared memory) show that accessing memory is not significantly different from just executing instructions. However, the communication does impose some software overhead. If the processors are communicating through some shared bus (UART, SPI, or just something as simple as parallel communication through the I/O ports), the overhead is equal to the cost of an interrupt (in our application, this was measured to be 130 cycles). In the multiprocesor design we modeled, the communication between the processors takes place over a UART, and costs one interrupt per byte, or 130 cycles per byte.

An analysis of the application running on a single processor revealed that 95% of the time is spent handling network events. The logical place to split the processing is somewhere within the network stack. Within the network stack itself most of the processor time is spent within the bit-level component (60% - handling the hardware interrupt, checking the internal state, reading and wrinting pins) and within managing the logic of byte-level transfers (30% - encoding and decoding of the raw bits, and controlling the lower level component). While it may seem attractive to split off the bit-level processing, since it consumes roughly half of the CPU time, it turns out to be a bad idea. Bit level interfaces are quite expensive to handle on traditional CPUs: the granularity of operations is quite mismatched between the incoming signal and the instructions operating on that signal. With our communication model, such split does not enable us to lower the frequency of any processors; in fact, since sending bytes over the UART is more expensive (in instructions that need to be executed) than a procedure call, we need to raise the frequency in order to meet the deadlines. It is marginally beneficial to to split off the combination of the bit-level and byte level radio interface. This a somewhat counterintuitive decision: after all these two components consume 90% of the CPU, but this split slightly reduces the maximum time it takes to process the incoming events: we are able to cut about 45 cycles from the maximum execution time of the events. However, this small reduction in required clock rate on the "radio processor" is somewhat offset by the requirements on the "main processor", which now needs to handle an additional 130 cycles of communications every 6400 cycles.

The initial evaluation of the multiple processor system seemed very discouraging. In fact, it seemed that the systems with multiple dedicated processors do not produce any power saving benefits. However, a careful examination of the application lead us to redesign some of the TinyOS control structures to produce significant power savings.

First, we note that almost 40% of the time critical tasks is devoted to saving state. These tasks run within interrupt handlers, and each invocation of an interrupt costs 60 cycles. This costs is inherent in the

TinyOS execution model: we allow events to interrupt threads, and that implies that an event handler has to save the processor state. In the partitioned application, the code running on a dedicated radio controller is well known and understood: it has a very regular control structure, a well understood interaction between threads and events, and threads with tightly bounded execution times. Given these properties, it is possible for us to rewrite the struture of the component so that there is only one context, and there is no need to save state on every event. For this project, we have simply reorganized the code manually, in principle such transformations should be possible to automate. The new structure contains only a single execution context. The execution is time sliced: the code is effectively a finite state machine which goes through an atomic transition between two ticks of the clock. This not only allows us to remove the state saving overhead but also to eliminate the general purpose scheduler, which currently contributes 40 instructions which could have been spent sleeping.

With the restructured code, we were able to obtain about 40% savings in instruction counts over the original code. Worth noting is the fact that the communication costs within the restructured module were quite a bit lower: on each pass through the loop we chose to poll the UART rather than drive it through interrupts.

The restructuring of the code has the effect of reducing the overheads in the application. Running the entire application within the interrupt context eliminates the need to save state, which corresponds to the "cheap interrupts" case from Table 4. Furthermore, in this case we were able to eliminate completely the scheduling overhead, which enabled further energy savings. As Table 4 indicates, the elimination of these overheads reduces the energy requirements of the application by 29%, a significant improvement.

5.2 Frequency Scaling

In the previous Section we saw that the application running on multiple processors can in fact be more energy-efficient than on a single processor. Now, we try to extract exactly what features of multiple processor design contribute to the power savings.

The total current consumption is the sum over all N processors p of the current I_p each draws,

$$\begin{aligned} I_N &= \sum_{p \in \text{procs}} I_p \\ &= NI_I + (\sum_{p \in \text{procs}} \rho_p) I_A \\ &= NI_I + (\frac{1}{fT} \sum_{p \in \text{procs}} K_p) I_A \\ &= NI_I + (\frac{K}{fT}) I_A. \end{aligned} \tag{10}$$

The K term includes cycles spent controlling devices across all processors. As before, this includes both overhead and essential work. The core work will be very similar in both single- and multiple-processor implementations. We assume a shared memory model and hold the cycles spent performing work, K_w, the same for both cases.

To keep the analysis simple, we also defer synchronization issues to the scheduling system, assuming it is perfect. This seems unrealistic, but many initial applications for simple networked sensors do not need heavy synchronization. One processor can process its measurements and write them to a slot in memory; another can read from that slot periodically and communicate what it finds. If the processing runs for a bounded time and is triggered by a real-time event, then the results are valid at another real time, much like the establishment of output signal in a circuit. The cascading real-time constraints increase the scheduling complexity, but the scheduling remains reasonable if the initial application is simple.

If using N processors lowers the necessary execution frequency to f/N, when will moving from one processor to N processors give us an energy savings? Let the total work plus the total overhead be the same for both one and N processors. The single processor has the same utilization ρ as above. Using ρ for K/fT in the multiple processor current will give comparisons from the single processor viewpoint. It is important to remember that ρ depends inversely on f, so scaling f by $1/N$, say, will scale ρ by N. We will continue to use f_m in the following sections for the constant value of $\rho f = K/T$ with the same caveats.

We want to know when $I(f) > I_N(f/N)$. Expanding $I_N(f/N)$ yields

$$I_N(f/N) = a_I f + N b_I + a_A f_m + N b_A \rho, \tag{11}$$

giving the unfortunate result that $I(f) > I_N(f/N)$ is true only when

$$(1 - N)(b_I + b_A \rho) > 0. \tag{12}$$

Physically, both b_I and b_A must be non-negative, and N is an integer greater than one, so this is *never* possible. It may be possible to scale frequency more rapidly than $1/N$. Using multiple processors reduces the real-time scheduling pressures and can result in a drastic contraction in frequency. The frequency scaling necessary for energy savings can be found by examining the inequality $I_N(g(f,N)\cdot f) < I(f)$, but we yet to extract from it a simple, meaningful result.

5.3 Frequency Scaling and Overhead Reduction

Is it possible to decrease the total current even with a simple frequency scaling? While lowering the necessary frequency decreases the active and idle currents at a given voltage, we have also seen that increasing the frequency may decrease the total current necessary by eliminating overhead cycles. Multiple processors can allow both frequency reduction and overhead elimination. Previously we had to increase execution frequency to remove context-switching overhead, but now we can consider handling multiple contexts through processor allocation. We can also simplify or remove the scheduler, reducing the overhead cycles even further.

Again, separate the work and overhead cycles as $K = K_w + K_o$ for a single processor utilization of $\rho = \rho_w + \rho_o$. Now assume that we can remove the overhead cycles by using N processors while simultaneously scaling the frequency by $1/N$. This gives a total current of

$$I'_N(f/N) = a_I f + N b_I + a_A f \rho_w + N b_A \rho_w. \quad (13)$$

Determining when $I'_N(f/N) < I(f)$ is equivalent to the test

$$(N-1)(b_I + b_A\rho_w) < I_A(f)\rho_o. \quad (14)$$

The left side, $(N-1)(b_I+b_A\rho_w)$, is the current drawn by the additional processors. The right, $I_A(f)\rho_o$, is the current drawn from the overhead cycles. If the frequency scales with $1/N$, then the tasks can be partitioned over as many as

$$N < 1 + \frac{I_A(f)\rho_o}{b_I + b_A\rho_w} \quad (15)$$

processors while using less current.

Note that scaling the frequency by $1/N$ kept the a_I terms equal, eliminating the primary difference between a processor's stop and idle modes from the comparison. However, if the stop mode requires a few cycles to restart fully, they can affect the threshold. The restart overheads will occur in the places where context switches were not necessary, when an interrupt-driven event is triggered and nothing else is active. These may occur more often in the multiple processor case. The restart cycle counts will be needed when examining a particular application. Once obtained, adding the different restart cycles, K_r and $K_{r,N}$, as overheads to *both* cases is equivalent to adding $K_r - K_{r,N}$ to K_o and $K_{r,N}$ to K_w in the previous inequalities. If $K_o + K_r - K_{r,N} < 0$, the multiprocessor restart cycles dominate, and the right side of Equation 14 is negative. Using more processors with only a $1/N$ scaling consumes more energy. One can examine applying a general scaling $g(f,N)$ to f, but we have yet to fully explore those results.

But what if various scheduling constraints conspire to disallow frequency scaling? The comparison of the additional current for the processors to that for the overhead gives

$$(N-1)I_I(f) < I_A(f)\rho_o \quad (16)$$

for $I'_N(f) < I(f)$. So it may still be a win, especially if the system uses the processor's stop mode.

5.4 Multiple Execution Units and Contexts

We have demonstrated that spreading tasks across multiple processors can decrease the power consumed by a networked sensor. In the derivation of Equation 10, we assumed that the idle currents of multiple processors accumulated into NI_I. This models the current needs of multiple physical processor chips, each pulling I_I over all T. What if they could all share the same idle current? Then the total current drawn by the N execution units is

$$I''_N = I_I + I_A\rho_w. \quad (17)$$

This consumes $I_A\rho_o$ less current than the simple single processor at the same frequency f. The frequency scalings that give energy savings can be bounded by solving the appropriate quadratic from $I(f) - I''_N(g(f,N)f)$.

Multiple execution units feeding from the same idle current provide an easy way to eliminate overhead. Our assumption that the total work cycles stayed the same when partitoned over the units provides the same result with a single execution unit that supports multiple hardware contexts. Multiple hardware

contexts and threaded processors give performance boosts in general purpose computers [15], and they also appear to give energy savings in small, event-driven sensors. Similar advantages come from data-flow designs and register partitioning.

6 Related Work

Lately, energy consumption has been a hot topic.

Shin and Choi [14] have adapted a fixed-priority, pre-emptive, real-time scheduler for low power devices. They demonstrate a significant power reduction for complex applications on a larger system that can vary both frequency and voltage dynamically while instructions are executed. They slow the processor down when only one task is available for execution, greedily stretching to 100% utilization. Their model includes the time for frequency and voltage changes but preemption overhead.

A wide variety of techniques for lowering power consumption are surveyed by Lorch and Smith [8]. While their primary focus is on general-purpose portable computers, they also discuss interaction with wireless communication and other I/O devices. They cover power control interactions between components in a user-oriented system, where we focus strictly on the processor in a sensor network. We hope that the understanding gained from our focus can contribute to a needed whole-system perspective for networked sensors.

Henkel examines the power implications of hardware-software partitioning in [5]. His method finds high-energy pieces of software and moves them into dedicated ASIC cores rather than adding additional power controls to the main processor's hardware. The ASICs not only consume less energy for the same function but also perform it more quickly. The speed reduces overhead concerns, giving an additional boost. He finds both faster and more efficient execution in a variety of problems, although his particular algorithm does produce a large slow-down for one application.

Kirovski and Potkonjak develop an algorithm for power-conscious task partitioning in a hard real-time system [7]. Their applications, information streaming, and task model are similar to ours, but they use heterogenous processors and work at a level where the frequency is determined by internal CMOS delay bounds. Also, they give each real-time task a dedicated processor, eliminating all context-switching overhead from the beginning.

7 Conclusions

We have presented the tradeoffs between the single- and multiprocessor designs for networked sensors. Neither design automatically creates a power efficient system. We have shown that it is possible to save energy while running on multiple processors; we have also shown that this is not true for mere partitioning of an existing application. The main observation from the study is that we do not save energy by simple frequency scaling, but rather from the elimination of various types of overhead. The avoidable overhead is very strongly dependent on a particular architecture. In a single processor system, most of the overhead comes from task switching, in the form of either scheduling overhead, or in the form of context saving. In the multiple processor design, the software overhead is introduced by the costs of extra communication. It is not the multiple ALUs or the multiple memories that produce the energy savings, but rather the simplified control structure, and the ability to eliminate context switches. The optimal architecture for networked sensors should combine the best features of both design types, by building upon a single processor with hardware support for multiple contexts. Architectures like SPARC with its register windows or PIC with a very large register set might be very appropriate for the networked sensors.

References

[1] Association for Computing Machinery. *Proceedings of the 36th ACM/IEEE conference on design automation*. ACM Press, June 1999.

[2] Atmel, Inc. AT90S4434/LS4434/S8535/LS8535 *Preliminary (Complete) Datasheet*.

[3] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard real-time scheduling: The deadline monotonic approach. In *Proceedings of the 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, May 1991.

[4] Kevin Bagett. Avr simulator and disassembler. `http://members.xoom.com/kb_badgett/`, 2000.

[5] Jörg Henkel. A low power hardware/software partitioning approach for core-based embedded systems. In *Proceedings of the 36th ACM/IEEE conference on design automation* [1], pages 122–127.

[6] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, Kristofer Pister. System architecture directions for networked sensors, 2000.

[7] D. Kirovski and M. Potkonjak. System-level synthesis of low-power hard real-time systems. In *Proceedings of the 34th ACM/IEEE conference on design automation*, pages 697–702. Acm Press, June 1997.

[8] Jacob Lorch and Alan Smith. Software strategies for portable computer energy management. *IEEE Personal Communications Magazine*, 5(3):60–73, June 1998.

[9] James McLurkin. Algorithms for distributed sensor networks. In *Masters Thesis for Electrical Engineering at the Univeristy of California, Berkeley*, December 1999.

[10] Massoud Pedram and Qing Wu. Design considerations for battery-powered electronics. In *Proceedings of the 36th ACM/IEEE conference on design automation* [1], pages 861–866.

[11] K. S. J. Pister, J. M. Kahn, and B. E. Boser. Smart dust: Wireless networks of millimeter-scale sensor nodes, 1999.

[12] RF Monolithics. Tr*1000 916* Mh*z Hybrid Transceiver*.

[13] Philips Semiconductors. The I^2C-bus specification, version 2.1. `http://www-us.semiconductors.com/acrobat/various/I2C_BUS_SPECIFICATION_\%3.pdf`, 2000.

[14] Youngsoo Shin and Kiyoung Choi. Power conscious fixed priority scheduling for hard real-time systems. In *Proceedings of the 36th ACM/IEEE conference on design automation* [1], pages 134–139.

[15] Radhika Thekkath and Susan J. Eggers. The effectiveness of multiple hardware contexts. In *Proceedings of the 6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 328–337. Acm Press, October 1994.