

Energy Implications of Network Sensor Designs

Robert Szewczyk and Andras Ferencz

ABSTRACT

Technological progress in integrated, low-power, CMOS communication devices and sensors makes a rich design space of networked sensors viable. They can be deeply embedded in the physical world or spread throughout our environment. The fundamental constraint on these devices is energy consumption, as the largest and most expensive component is the battery. In this paper, we analyze the power usage of a remote networked sensor and evaluate the power implications of various design choices.

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 [7].

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?

As an initial exploration of this topic, we compare two networked sensor systems: 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.

To a first order, the power dissipation within a system is

$$P = CV^2 f$$

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. There is also an effect similar to “contraction” effect in chemistry: if two processors need to run at frequencies f_1 and f_2 in order to meet their deadlines, a single processor might be able to run at a frequency lower than $f_1 + f_2$ in order to meet the deadlines for both tasks, since the case of simultaneous deadline pressure might never occur.

Here, we explore some aspects of this problem based on a specific instance of a networked sensor. The rest of this paper is organized as follows. Section 2 discusses the details of the hardware and software system under evaluation. In order to evaluate a sample multi-processor design, we present a set of detailed energy measurements of the single processor design in Section 3. From this data, we develop a power aware simulator that can emulate a multi-processor sensor, and with the help of a test application to evaluate the design points in Section 4. Finally, Section 5 concludes.

2. EXPERIMENTAL PLATFORM

In order to illustrate the tradeoffs between the competing designs, we examine a simple application for a networked sensor developed in [5]: 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 [6] using the TinyOS framework [5]. In this section we briefly present the hardware.

2.1 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 Monolithic 916.50 MHz transceiver (TR1000) [3], 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 [1] externally clocked at 4MHz. It is an 8-bit Harvard 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 IO 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 milliseconds to restore the processor from the deeper sleep modes.

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 I²C [8] protocol is used with software on the microcontroller synthesizing the I²C master over general I/O pins.

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

2.2 TinyOS

As a model of execution for the network sensor application, we chose the TinyOS described in [5]. 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 considerably more programmable.

A complete TinyOS system consists of a simple scheduler, and a graph of *components*. Each component has four inter-related 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 asynchronously 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 I²C, 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 of the components 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.

2.3 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 decoded¹ 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. 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.

The temperature sensor uses the I²C 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 IO 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.

3. MEASUREMENTS

Our measurements aim to facilitate the creation of a power-aware mote simulator (see Section 4). Thus we are interested in producing energy consumption values for each processor instruction and models for the external devices.

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

3.1 Experimental Setup

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.

3.2 Micro Benchmarks

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 is:

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

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

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.
RMF send 100 s pulse	2.56	2050 nJ/bit
RFM receive	2.44	1950 nJ/bit

Table 1: Energy consumption of instructions (left) and external modules (right).

Since the light sensor is a photoresistor, the current drawn depends on the light level. The last two measurements 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.

3.3 Measurement Results

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 AT-MEL AVR architecture reveals that the register file, the IO 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 conclusions in Section 4.3.

4. ARCHITECTURAL EXPERIMENTS

4.1 Simulator Design

In order to model the energy consumption of the various SmartDust designs, we have developed a simulator for the AT-MEL processor, and integrated it with the basic peripherals, like timers, and basic IO ports.

Originally we imagined that in order to precisely model the energy usage of the smartdust mote, we would need precise energy per instruction data. Thus our simulator was designed to keep track of energy expended during the op-

Task	Average time (cycles)	Max. Time (cycles)	Period (μ s)
start symbol search	130	144	50
receive bit	191	315	100
send bit	163	301	100
signal encode	130	130	1600
signal decode	146	146	1600

Table 2: The timing requirements of real-time components of the system. The code for handling IO devices like the temperature controller does not require real-time guarantees, so it is not crucial in determination of the frequency of a particular component.

eration of the device, based on the instruction stream executed, the current state of the peripherals, and the sleep cycle. However as we pointed out in Section 3, all types of instructions seem to take similar amount of energy per cycle. Similarly, the measurements of integrated peripherals proved somewhat disappointing: we observed no difference in energy caused by UART communication or by changing the state of individual IO pins². This led to a conclusion that the energy accounting structure was unnecessary and the relevant parameters are the number of cycles spent in execution and the number of cycles spent in power down or sleep modes.

4.2 Single Processor Evaluation

Initially, the simulator was used to generate instruction traces for the single processor system in order to precisely profile the application. This experiment had some overlap with the measurements presented in [5]. While there may have been some duplication of effort, it increased our confidence to find that the measurements done using a logic analyzer match exactly the data obtained from simulation. Table 2 shows the timing requirements of the time-critical pieces of TinyOS. Note that the first three types of activity (start symbol search, receiving a bit, and sending a bit) are mutually exclusive, and that byte-level activities impose a much more lax requirements on the application.

Based on these figures, it appears that TinyOS actively looking for an incoming transmission pushes the limits of the hardware. If TinyOS were purely event based, then the processor would be 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 figure out the average time spent sleeping.

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

²Note that the discrete IO devices do consume quite substantial amounts of power; however in this study we are concentrating on the impact of multiple processors on the energy usage, and we are keeping other parameters constant.

times.

4.3 Multiple processor designs

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 substantial freedom in choosing the partitioning of components. Our measurements show that the communication between two processor over an external bus is essentially free (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 200 cycles). In the multiprocessor design we modeled, the communication between the processors takes place over a UART, and costs one interrupt per byte, or 200 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 writing 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 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 200 cycles 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 structure 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.

This experiment yielded significant power benefits, yet they were not intrinsic to a multiple processor design. In particular, similar benefits could be obtained with a single multi-threaded processor, like one used in Terra processors, or with an architecture which could provide a very low cost context switch, like Sparc with register windows ([4]) or Microchip PIC with a very large number of registers ([2]).

4.4 Frequency and Energy

So far, the analysis of the energy usage revolved around the frequency required to meet the real time deadlines. However, the frequency affects the *power*, rather than energy. Figure 1 shows the relationship between the processor frequency and the current drawn. For a given task executing on a processor, the energy dissipated is the sum of energy dissipated while active and the energy dissipated while idle. If the idle energy depends linearly on frequency (like in the ATMEL case), then there is an optimal frequency for running an application: this frequency minimizes the number of idle cycles³. The system running the same instruction sequences will spend similar amounts of energy performing useful work regardless of the frequency or partitioning among the component processors. The energy usage differences will come mainly from two sources: overheads associated with the particular arrangement of functionality, and from the idle time. It is through the minimization of these factors that we achieve an energy efficient system.

³We derive this result in a different class project, which is not yet written up. Interestingly enough, if the idle current is constant w.r.t. frequency, then processors should be ran as fast as possible, since bursty utilization would lead to a lower overall power usage

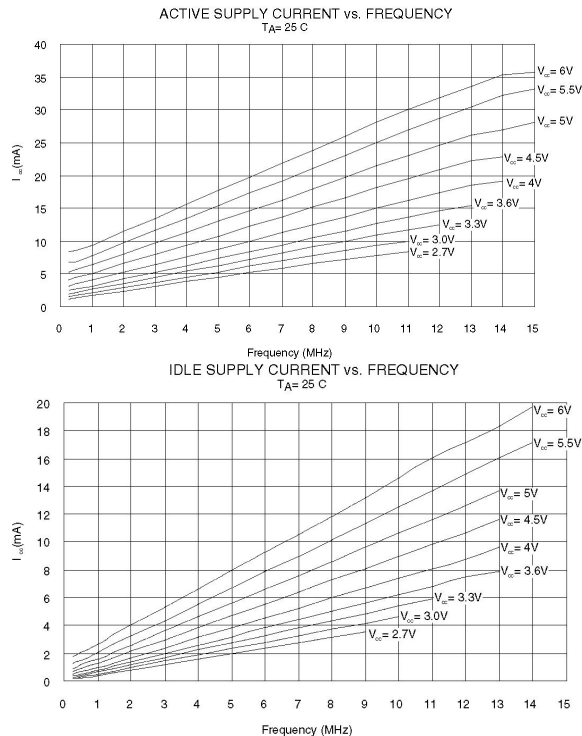


Figure 1: Current supply charts for the AT90LS8535. Note that in most of the space presented, the both idle and active currents are almost exactly linear with frequency. Also note that the idle mode is quite power hungry: when idle, the MCU uses almost 50% of active power.

5. CONCLUSION

In a system with real-time constraints, the processor is idle because of non-uniformity of task execution times: the system must be fast enough to handle the worst case execution time, and the idle time is determined by the difference between the worst case and average execution times. If an application is split in such a way that the idle time is minimized, then it can achieve its optimal power usage. We were unable to observe a significant energy reduction caused by the mere ability to reduce the clock speed of the processor without missing deadlines: in our case, the potential benefit was eliminated by the cost of exchanging data between independent units. A significant side benefit of partitioning is the ability to simplify the control structures running on individual processors, and thus reduce the overhead required for multiple processes to coexist. It is through the assignment of tasks to individual execution contexts that we were able to show power savings. However, this gain can be just as easily realized on a single processor with cheap context switches. The next generation of networked sensors should recognize the highly concurrent nature of the environment by providing efficient ways of switching between lightweight tasks.

6. REFERENCES

- [1] Atmel AVR 8-Bit RISC processor. <http://www.atmel.com/atmel/products/prod23.htm>.
- [2] PICmicro Reference Manual. <http://www.microchip.com/10/Lit/PICmicro/Refernce/index.htm>.
- [3] RF Monolithics. <http://www.rfm.com/products/data/tr1000.pdf>.
- [4] Anant Agarwal, Geoffrey D'Souza, Kirk Johnson, David Kranz, John Kubiawicz, Kiyoshi Kurihara, Beng-Hong Lim, Gino Maa, Daniel Nussbaum, Mike Parkin, and Donald Yeung. The MIT alewife machine : A large-scale distributed-memory multiprocessor. In *Proceedings of Workshop on Scalable Shared Memory Multiprocessors*. Kluwer Academic, 1991.
- [5] Jason Hill, Robert Szweczyk, Alec Woo, Seth Hollar, David Culler, Kristofer Pister. System architecture directions for networked sensors, 2000.
- [6] James McLurkin. Algorithms for distributed sensor networks. In *Masters Thesis for Electrical Engineering at the Univeristy of California, Berkeley*, December 1999.
- [7] K. S. J. Pister, J. M. Kahn, and B. E. Boser. Smart dust: Wireless networks of millimeter-scale sensor nodes, 1999.
- [8] Philips Semiconductors. The i²c-bus specification, version 2.1. http://www-us.semiconductors.com/acrobat/various/I2C_BUS_SPECIFICATION_\\%3.pdf, 2000.