Datalogisk Institut
KØBENHAVNS UNIVERSITET

Characterizing Mote Performance: A Vector-Based Methodology

Martin Leopold, Marcus Chang, Philippe Bonnet

Technical Report no. 07/06
ISSN: 0107-8283

Dept. of Computer Science
University of Copenhagen • Universitetsparken 1
DK-2100 Copenhagen • Denmark

Characterizing Mote Performance: A Vector-Based Methodology

Authors removed for Double-Blind Reviewing

Institution removed for Double-Blind Reviewing

Abstract. Sensors networks instrument the physical space using motes that run network embedded programs thus acquiring, processing, storing and transmitting sensor data. The motes commercially available today are large, costly and trade performance for flexibility and ease of programming. A new generation of motes, designed as System-on-a-Chip, is promising to deliver a significant improvement in terms of power consumption and price, possibly at the cost of reduced CPU efficiency. How to compare mote performance? How to find out which mote is best suited for a given application. In this paper, we propose a vector-based methodology for benchmarking mote performance. Our method is based on the hypothesis that mote performance can be expressed as the scalar product of two vectors, one representing the mote characteristics, and the other representing the application characteristics. We implemented our approach in TinyOS 2.0 and we present the result of experiments obtained on commercial motes from Sensinode. We give a quantitative comparison of these motes, and predict the performance of a data acquisition application.

1 Introduction

Sensor networks-based monitoring applications range from simple data gathering, to complex Internet-based information systems. Either way, the physical space is instrumented with sensors extended with storage, computation and communication capabilities, the so-called motes. Motes run the network embedded programs that mainly sleep, and occasionally acquire, communicate, store and process data. In order to increase reliability and reduce complexity, research prototypes [1, 2] as well as commercial systems[1] now implement a tiered approach where motes run simple, standard data acquisition programs while complex services are implemented on gateways. The data acquisition programs are either a black box (Arch Rock), or the straightforward composition of building blocks such as sample, compress, store, route (Tenet). This approach increases reliability because the generic programs are carefully engineered, and reused across deployments. This approach reduces complexity because a system integrator does not need to write embedded programs to deploy a sensor network application.

Such programs need to be portable to accommodate different types of motes. First, a program might need to be ported to successive generations of motes.

[1] See http://www.archrock.com

Indeed, hardware designers continuously strive to develop new motes that are cheaper, and more power efficient. Second, a program might need to be ported simultaneously to different types of motes, as system integrators need various form factors or performance characteristics.

Handzicki, Polastre et al.[5] address the issue of portability when they designed TinyOS 2.0 Hardware Abstraction Architecture. They defined a general design principle, that introduces three layers:

1. Mote Hardware: a collection of interconnected hardware components (typically MCU, flash, sensors, radio).
2. Mote Drivers: Hardware-specific software that exports a hardware independent abstraction (e.g., TinyOS 2.0 define such Hardware Independent Layer for the typical components of a mote).
3. Cross-Platform Programs: the generic data acquisition programs that organize sampling, storage and communication.

Whether motes are deployed for a limited period of time in the context of a specific application (e.g., a scientific experiment), or in the context of a permanent infrastructure (e.g., within a building), power consumption is the key performance metric. Motes should support data acquisition programs functionalities within a limited power budget. We focus on the following questions:

1. What mote hardware to pick for a given program? The problem is to explore the design space and choose the most appropriate hardware for a given program without having to actually benchmark the program on all candidate platforms.
2. What is a mote hardware good for? The problem is to characterize the type of program that is well supported by a given mote hardware.
3. Is a driver implemented efficiently on a given hardware? The problem is to conduct a sanity check to control that a program performs as expected on a given hardware.

We are facing these questions in the context of the Snowths project, where we design a data acquisition infrastructure. First, because of form factor and cost, we are considering a System-on-a-Chip as mote hardware. Specifically, we want to investigate whether Sensinode Nano, a mote based on Chipcon's CC2430 System-on-a-Chip, would be appropriate for our application. More generally, we want to find out what a CC2430 mote is good for, i.e., what type of applications it supports or does not support well. Also, we had to rewrite all drivers to TinyOS 2.0 on CC2430, and we should check that our implementation performs as well as TinyOS 2.0 core. Finally, we would like to use Sensinode Micro (a mote quite similar to Telos) as a prototyping platform for our application as its toolchain is easier and cheaper to use (see Section 3.2 for details). We would like to run our application on the Micro, measure performance, and predict the performance we would get with the Nano.

Typically, analytical models, simulation or benchmarking are used to study the performance of a program [3]. In our opinion, simulation is best suited for

reasoning about the performance and scalability of protocols and algorithms, not to reason about the performance of an application program on a given mote hardware. Indeed, simulators are best suited when they abstract the details of the hardware and driver layers. Standard benchmarks fall into two categories: application benchmarks (SPEC, TPC), or microbenchmarks (lmbench)[2]. There is no such standard benchmark for sensor networks. Micro benchmarks have been defined for embedded systems (EEMBC), but they focus at the automotive and consumer electronics markets – they do not tackle wireless networking or sensing issues.

In this paper, we propose a vector-based methodology to study mote performance. Our hypothesis is that energy consumption on a mote can be expressed as the scalar product of two performance vectors, one that characterize the mote (hardware and drivers), and one that characterize the cross-platform application. Using this methodology, we can compare motes or applications by comparing their performance vectors. We can also predict the performance of an application on a range of platforms using their performance vectors. Specifically, our contribution is the following:

1. We adapt the vector-based methodology, initially proposed by Seltzer et al.[4], to study mote performance in general and TinyOS-based motes in particular.
2. We conduct experiments with two types of motes running TinyOS 2.0: Sensinode Micro and CC2430. We ported TinyOS to these platforms. The Micro is very similar to Telos, so porting was straightforward. CC2430 is a System-on-a-Chip, based on a 8051 micro-controller; it is quite different from the core platforms supported by TinyOS. We detail our implementation of TinyOS on CC2430 in Section 3.
3. We present the results of our experiments. First, we test the hypothesis underlying our approach. Second, we compare the performance of the Micro and CC2430 motes using their hardware vectors. Finally, we predict the performance of generic data acquisition programs from the Micro to the CC2430.

2 Vector-Based Methodology

The vector-based methodology, initially proposed by Seltzer et al.[4], consists in expressing overall system performance as the scalar product of two vectors:

1. A system-characterization vector, which we call **mote vector** and denote $\underline{MV}$. Each component of this vector represents the performance of one primitive operation exported by the system, and is obtained by running an appropriate microbenchmark.

[2] See http://www.tpc.org, http://www.spec.org, http://www.bitmover.com/lmbench, and http://www.eembc.org/ for details about these benchmarks.

2. An application-characterization vector, which we call **application vector** and denote $\underline{AV}$. Each component of this vector represents the application's utilization of the corresponding system primitives, and is obtained by instrumenting the API to the system primitive operations.

Our hypothesis is that we can define those vectors such that mote performance can be expressed as their scalar product:

$$Energy = \underline{MV} \cdot \underline{AV}$$

Our challenge is to devise a methodology adapted to mote performance. The issues are (i) to define the mote vector components, and the microbenchmarks used to populate them, and (ii) to define a representative application workload, to collect a trace from the instrumented system API, and to convert an application trace into an application vector.

2.1 Mote Vector

We consider a system composed of the mote hardware together with the mote drivers. The primitive operations exported by such a system are:

- CPU duty cycling: the network embedded programs that mainly sleep and process events need to turn the CPU on and off[3].
- Peripheral units: controlled through the hardware-independent functions made available at the drivers interface.

We choose this system because its interface is platform-independent. This has two positive consequences. First, we can use mote vectors to compare two different motes. Second, the application vector is platform-independent. We can thus use our vector-based methodology to predict the performance of an application across motes.

The mote vector components correspond to the CPU (when active or idle), and the peripheral units (as determined by the driver interfaces). Throughout the paper, we use an associative array notation to denote the mote (and application) vector components, e.g., $\underline{MV}[active]$ corresponds to CPU execution, $\underline{MV}[idle]$ corresponds to CPU sleep, $\underline{MV}[PUi]$, correspond to peripheral units primitives where PUi is for example ADC sample, flash read, flash write, flash erase, radio transmit, radio receive.

We need to define a metric for the vector components. The two candidates are energy and time. We actually need both: (a) energy to compute the scalar product with the application vector and thus obtain mote performance, and (b) time to derive the platform-independent characteristics of an application (see Section 2.2). We thus need to define a microbenchmark for each mote vector

[3] Note that we assume that the mote hardware relies on a single CPU to control all peripheral units. Peripheral units such as digital sensors might include their own micro-controller. Our assumption simply states that a mote program is run on a single CPU.

component for which we measure time elapsed and energy spent. We distinguish between the energy mote vector, noted $\underline{MV_e}$, and the time mote vector, noted $\underline{MV_t}$.

The microbenchmarks must capture the performance of the system's primitive operations. The first problem is to represent CPU performance. The most formidable task for the CPU in a sensor network application is to sleep. This is why we distinguish sleep mode from executing mode in the mote vector. For the applications we consider, a single sleep mode is sufficient. Defining a microbenchmark to define the energy spent in sleep mode is trivial. However, we wish to use the time mote vector to compare the time spent in sleep mode by different motes. Intuitively, the time spent in sleep mode is a complement of the time spent processing. As an approximation, we thus consider that $\underline{MV_t}[idle]$ is the complement of $\underline{MV_t}[active]$ with respect to an arbitrary time period (fixed for all mote vectors), and that $\underline{MV_e}[CPUsleep]$ corresponds to the energy spent in sleep mode during that time.

The second problem is to define an appropriate representation of CPU performance (in executing mode). Unlike peripheral units, for which drivers define a narrow-interface, the CPU has a rich instruction set. It is non-trivial to estimate the CPU resources used by a given application as it depends on the source code and on the way the compiler leverages the CPU instruction set. We choose a simple approach where we use a microbenchmark as a yardstick for the compute-intensive tasks of an application. We thus represent CPU performance using a single vector component. There is an obvious pitfall with this approach: we assume that the distribution of instructions used by the microbenchmark is representative of the instructions used by the application. This is unlikely to be the case. We use this simple approach, despite its limitation, as a baseline for our methodology because we do not expect CPU utilization to have a major impact on energy consumption. Our experiments constitute a first test of this assumption. Obviously much more tests are needed, and devising a more precise estimation of CPU utilization is future work.

The third problem related to the microbenchmarks is that driver interfaces often provide a wide range of parameters that affect their duration and energy consumption. Instead of attempting to model the complete range of parameters, we define microbenchmarks that fix a single set of parameters for each peripheral unit primitive. Each peripheral unit microbenchmark thus corresponds to calling a system primitive with a fixed set of parameters, e.g., a microbenchmark for radio transmit will send a packet of fixed length, and a microbenchmark for ADC sampling will sample once at a fixed resolution. We believe that this models the behavior of sensor network application that typically use a fixed radio packet length or a particular ADC resolution. This method can trivially be expanded by defining a vector component per parameters (e.g., replacing radio transmit with two components radio transmit at packet *length_1* and radio transmit at packet *length_2*).

For the sake of illustration, let us consider a simplistic mote with a subset of the TinyOS 2.0 drivers, that only exports two primitives: ADC sample and

radio transmit (tx). The associated time mote vectors will be of the form:

$$\underline{MV_t} = \begin{bmatrix} active \\ idle \\ adc \\ tx \end{bmatrix}$$

Where the mote vector components correspond to the time spent by the mote running the CPU microbenchmark, to the time spent in sleep mode (the complement of the time spent running the CPU benchmark with respect to an arbitrary time period that we set to 20 s), to the time spent running the ADC benchmark, and to the time spent running the transmit benchmark.

In order to express mote performance as the scalar product of the energy mote vector and the application vector, we need the components of the mote vectors to be independent. This is an issue here, because CPU is involved whenever peripheral units are activated. Our solution is to factor CPU usage in each peripheral unit component. As a consequence, the mote vector component corresponding to CPU performance (*active*) must be obtained without interference from the peripheral units. Another consequence is that we need to separate the CPU utilization associated to peripheral units from the pure computation, when deriving the platform-independent characteristics of an application. We thus register CPU time when benchmarking each peripheral unit primitive. We denote them as $CPU[PUi]$ for each peripheral unit primitive PUi.

We detail in the next Section, how we use those measurements when deriving the application vector from a trace.

2.2 Application Vector

Our goal is to characterize how an application utilizes the primitives provided by the underlying system. The first issue is to define a workload that is representative of the application. In the context of sensor networks, workload characterization is complicated (i) because motes interact with the physical world and (ii) because the network load on a mote depends on its placement with respect to the gateway, and (iii) because different motes play different roles in the sensor network (e.g., in a multihop network a mote located near the gateway deals with more network traffic than a mote located at the periphery of the network).

We consider that a sensor network application can be divided into representative epochs that are repeated throughout the application lifetime. For example, the application we consider in the Snowths project consists of one data acquisition epoch[4], where an accelerometer is sampled at 4 Hz, the samples are compressed, stored on flash when a page is full, and transmitted to the gateway

[4] A sensor network deployed for collaborative event detection will typically consist of two epochs: one where motes are sampling a sensor and looking for a given pattern in the local signal, and one where motes are communicating once a potential event has been detected.

when the flash is half-full. While sampling is deterministic, such an epoch is non-deterministic as compressing, storing or transmitting depends on the data being collected, and on the transmission conditions. Obviously, tracing an application throughout several similar epochs will allow us to use statistics to characterize these non-deterministic variations.

For each epoch, we trace how the application uses the CPU and the peripheral units. More precisely the trace records the total time spent by the mote in each possible mote state, defined by the combination of active mote vector components (*active* that represents the compute-intensive operations, *idle* that represents the CPU in sleep mode, and PUi that represents a peripheral unit interface call). We thus represent the trace as a vector, denoted $\underline{T}$. $\underline{T}$ is of dimension 2^m, where m is the dimension of the mote vector. Some of the mote states will not be populated because they are mutually exclusive (e.g., *active* and *idle*), or because the driver interfaces prevent a given combination of active peripheral units.

Let us get back to the simple example we introduced in the previous section. The trace vector for an epoch will be of the form:

$$\underline{T} = \begin{bmatrix} active \\ idle \\ adc \\ tx \\ adc \ \& \ tx \\ active \ \& \ adc \\ active \ \& \ tx \\ active \ \& \ adc \ \& \ tx \end{bmatrix}$$

Now the problem is to transform, for each epoch, the trace vector into a platform-independent application vector. The application vector, denoted $\underline{AV}$, has same dimension m as the mote vector, and each application vector component corresponds to the utilization of the system resource as modeled in the mote vector. The application vector components have no unit, they correspond to the ratio between the total time a system primitive is used in an epoch, by the time spent by this system primitive in the appropriate microbenchmark (as recorded in the time mote vector $\underline{MV_t}$). Note that if the driver primitive is deterministic, then the ratio between the total time spent calling this primitive in an epoch and the microbenchmarking time is equal to the number of times this primitive has been called. However, drivers typically introduce non-determinism,because the scheduler is involved or because drivers embed control loops with side effects (e.g., radio transmission control that results in retransmissions).

We use a linear transformation to map the trace vector onto the application vector. This transformation can be described in three steps:

1. We use an **architecture matrix** to map the trace into a vector of dimension m, the **raw total time vector**, where each component correspond to the total utilization of the CPU and peripheral units. The architecture matrix encodes the definition of each state as the combination of active mote vector

components. Note that this combination depends on the architecture of the mote. For example, a SPI bus might be shared by the radio and the flash. In this case, the time spent in a state corresponding to radio transmission and flash write is spent either transmitting packets or writing on the flash (there is no overlap between these operations). We assume fair resource arbitration and consider that both components get half the time recorded in the trace. In case of overlap between operations, both get the total time recorded in the trace.

In our simplistic example, assuming that a SPI resource is shared between the radio and the ADC, the architecture matrix will be of the form:

$$\mathbf{AM} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & \frac{1}{2} & 1 & 0 & \frac{1}{2} \\ 0 & 0 & 0 & 1 & \frac{1}{2} & 0 & 1 & \frac{1}{2} \end{bmatrix}$$

2. We use a **CPU matrix** to factor out of the *active* component the time spent by the CPU controlling the peripheral units. The CPU matrix, of dimension $m \times m$, is diagonal except for the column corresponding to the *active* component. This column is defined as 1 on the diagonal, 0 for the *idle* component, and $-\underline{CPU}[k]/\underline{MV}[k]$ for all other components. When multiplying the total time vector with the CPU matrix, we obtain a **total time** vector where the *active* component corresponds solely to the compute-intensive portion of the application.

   Using again our running example, we have a CPU matrix of the form:

$$\mathbf{CPU} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{CPU[adc]}{\underline{MV_t}[adc]} & 0 & 1 & 0 \\ -\frac{CPU[tx]}{\underline{MV_t}[tx]} & 0 & 0 & 1 \end{bmatrix}$$

3. We use the time mote vector to derive the application vector. The basic idea is to express the application utilization of the system primitive as the ratio between total time per component, and the time spent running a benchmark. We define the inverse mote vector, $\underline{MV^{-1}}$, as a vector of dimension m where each component is the inverse of the time mote vector component (this inverse is always defined as the time mote vector components are always non zero). We define the application vector as the Hadamard product of total time vector with the inverse mote vector.

   With our running example, we obtain the equation:

$$\begin{bmatrix} totalactive/\underline{MV_t}[active] \\ totalidle/\underline{MV_t}[idle] \\ totaladc/\underline{MV_t}[adc] \\ totaltx/\underline{MV_t}[tx] \end{bmatrix} = \begin{bmatrix} totalactive \\ totalidle \\ totaladc \\ totaltx \end{bmatrix} \circ \begin{bmatrix} 1/\underline{MV_t}[active] \\ 1/\underline{MV_t}[idle] \\ 1/\underline{MV_t}[adc] \\ 1/\underline{MV_t}[tx] \end{bmatrix}$$

More generally, we derive the application vector from the trace vector using the following linear transformation:

$$\underline{AV} = (\mathbf{CPU} \times (\mathbf{AM} \times \underline{T})) \circ \underline{MV^{-1}}$$

And we obtain the mote performance as the scalar product of the application vector with the energy mote vector:

$$E = \underline{AV} \cdot \underline{MV_e}$$

3 Implementation in TinyOS 2.0

We applied our vector-based methodology to two motes: Sensinode Micro, a Telos-like mote, and CC2430, which is the basis for a new generation of commercial motes[5]. We ported TinyOS 2.0 on both platforms.

3.1 CC2430 and Sensinode Micro

Chipcon's CC2430 is a System-on-a-Chip (SoC)[6]. As such it has a small form factor (7x7 mm) and promises to be mass-produced at a lower price than complex boards. Motes built around the CC2430 might constitute an important step towards reducing the price of sensor networks. The CC2430 is composed of the 8051 MCU and a wide range of on-chip peripherals such as timers, bus controllers as well as internal flash, ADC and an 802.15.4 radio very similar to the CC2420. We run the system clock from a 32 MHz external oscillator, which is required for operating the radio. The CC2430 features 4 power modes controlling the peripherals and an *idle* state that keeps peripherals running, but stops the CPU core.

The Intel 8051 MCU architecture was designed in the early eighties as an 8 bit, CISC style processor with a Harvard architecture which(i.e., code and data are located in separate memory spaces). It is quite different from the platforms on which TinyOS has been implemented so far. The main peculiarity of the 8051 is that it defines memory spaces that differ in size, are addressed differently and vary in access time. Simply put, the 8051 defines a fast memory area limited to 256 bytes, and a slow memory area of 8 KiB. In addition to variables, the fast access area contains the program stack. This limits the program stack to less than 256 bytes depending on the amount of variables in this area. Commonly, activation records of functions are placed on the stack, thus potentially limiting the call depth critically. To circumvent this problem, the compiler places stack frames in the slow data area, which imposes a high cost for storing and retrieving arguments that do not fit in registers when calling a function. The slow access RAM also penalizes dynamic memory allocation, and context switches and thus favor an event-based OS with static memory allocation such as TinyOS.

[5] We experimented with a CC2430 development kit. Using commercial systems based on CC2430, such as Sensinode Nano, is future work.

[6] For details, see CC2430 data sheet: http://focus.ti.com/lit/ds/symlink/cc2430.pdf

Because CC2430 is a System-on-a-Chip, there is no bus between the MCU and the radio. The MCU controls the radio via special function registers (instead of relying on a SPI bus as it is the case on Telos and Micro motes for example). The other peripheral units (ADC, UART, timers, flash, and pins) are accessed in the 8051 MCU as in other micro-controllers such as the MSP or Atmega.

The Sensinode Micro is built around the MSP430 MCU and features a stackable design. The platform can run up to 8 MHz (we choose 1 MHz in our experiments). Apart from the built in peripherals of the MSP, it features the Texas Instruments CC2420 radio and an external flash. The flash and radio units are interconnected through a common bus by multiplexing the SPI.

3.2 TinyOS 2.0 on CC2430 and Micro

TinyOS 2 has been designed to facilitate the portability of applications across platforms. First, it is built using the concept of components that use and provide interfaces. TinyOS is written in nesC, an extension of C that supports components and their composition. TinyOS provides a set of system components (e.g. initialization, scheduler, etc.) a set of library components (e.g. random number generator, queue, etc.) and a set of platform specific components. Second, TinyOS implements the Hardware Abstraction Architecture[5]. For each hardware resource, a driver is organized in three layers: the Hardware Presentation Layer (HPL) that directly exposes the functions of the hardware component as simple function calls, the Hardware Abstraction Layer (HAL) that abstracts the raw hardware interface into a higher-level but still platform dependent abstraction (possibly maintaining state), and the Hardware Interface Layer (HIL) that exports a narrow, platform-independent interface.

The TinyOS 2.0 core working group has defined Hardware Interface Layers for the hardware resources of typical motes: radio, flash, timer, ADC, general IO pins, and UART[7]. Porting TinyOS 2.0 on CC2430 consisted in implementing these drivers:

Radio We export the radio using a straightforward *SimpleMac* interface. This interface is well suited for the 802.15.4 packet-based radios of the CC2430. It allows to send and receive packets, and set various 802.15.4 parameters (channel, transmission power, MAC and PAN addresses, filters) as well as duty cycling the radio (with rxEnable or rxDisable). Note that we depart from the Active Message abstraction promoted by the TinyOS 2.0 core working group. Our SimpleMac implementation supports simple packet transmission, but does not provide routing, or retransmission. Implementing Active Messages is future work.

Flash We export the flash using the *SimpleFlash* interface that allows to read and write an array of bytes, as well as delete a page from flash. Note that this interface is much simpler than the abstractions promoted by the TinyOS 2.0 core working group (volumes, logging, large and small objects). We adopted

[7] For details, see http://www.tinyos.net/scoop/special/working_group_tinyos_2-0

this simple interface because it fits the needs of our data acquisition application. Implementing the core abstractions as defined in TEP103 is future work.

Timer The timers are exported using the generic TinyOS Timer interfaces *Alarm* and *Counter*. These two interfaces give applications access to hardware counters and allows the use of the TinyOS components to extend the timer width from 16 bit to 32 bit. Note that on the pre-release CC2430 chips we used for our experiments, timers do not work properly[8].

ADC The Analog-to-Digital Converter is accessed through the core *Read* interface that allows to read a single value. In order to read multiple values, an application must issue multiple read calls or use DMA transfers.

Pins The General IO pins are exported through the core *GeneralIO* interface, that allows to set or clear a pin, make it an input or an output.

UART The UART is exported using the core *SerialByteComm* interface (that sends and receives single bytes from the UART) and *StdOut* interfaces (that provides a `printf`-like abstraction on top of SerialByteComm.

Note that we did not need to change the system components from TinyOS 2.0. However, supporting a sleep mode on the CC2430 requires implementing a low-frequency timer. This is work in progress, as a consequence our experiments are conducted without low-power mode on the CC2430.

The main challenges we faced implementing TinyOS 2.0 drivers on CC2430 were to (i) understand the TEP documents that describe the core interfaces as we were the first to port TinyOS 2.0 on a platform that was not part of the core, and (ii) to define an appropriate tool chain. Indeed, the code produced by the nesC pre-compiler is specific to gcc, which does not support 8051. We had to (a) choose another C compiler (Keil), and (b) introduce a C-to-C transformation step to map the C file that nesC outputs into a C file that Keil accepts as input (e.g., Keil does not support inlining, the definition of interrupt handlers is different in Keil and gcc, Keil introduces compiler hints that are specific to the 8051 memory model). The details of our toolchain are beyond the scope of this paper, see Reference [6] for details.

The Micro has many similarities with the Telos motes on which TinyOS 2.0 was originally developed. Thus porting most TinyOS 2.0 drivers features was a simple exercise. However, the wiring of the radio does not feature all of the signals available on the Telos mote, meaning that the radio stack could not be reused. We implemented the simple MAC layer, *SimpleMac*, and simple flash layer *SimpleFlash* described above.

3.3 Mote Vectors and Benchmarks

Both motes export the same driver interfaces. As a result, we chose the following components for their mote vectors: *active*, *idle*, *adc*, *radio_receive*, *radio_transmit*,

[8] The timers miss events once in a while. This error is documented on a ChipCon errata, which is not publically available.

flash_read, *flash_write*, and *flash_erase*. Doing so, we leave some of the peripheral unit primitives out of the mote vector (e.g., the primitives to set or get the channel on the 802.15.4 radio). This is because we factor in the time spent there as CPU execution. We also leave timers, UART and general IO pins out of the mote vector. The time spent in the timers is factored in the CPU idle component. We leave general IO pins out because we do not use LEDs, or digital sensors. Similarly, we do not use the UART. Note that we do not consider a specific sensor connected to the ADC. Also, the Micro has both internal and external flash (while the CC2430 has only internal flash). We report the results we got with the internal flash.

The benchmarks we defined for these mote vector components are:

- A compression algorithm to characterize CPU execution. This component contains a mix of integer arithmetic with many loads and stores and some function calls. Using this algorithm is a baseline approach.
- Simple function calls with a fixed parameter for each peripheral unit primitive[9]. Note that benchmarks, in particular for the radio and flash, contain some buffer manipulation. These are measured as $CPU[PUi]$ (see Section 2.1).

3.4 TinyOS API Instrumentation

We need to implement the CPU and peripheral units to collect the traces that are the basis for the application vectors. We implemented the following mechanisms:

- For the peripheral units, we introduce a platform-independent layer between the component that provides the driver interface and the component that uses it. As an example consider reading a value from the ADC using the TinyOS 2.0 *Read* interface. This interface starts an ADC conversion with a *Read* command and returns with a *readDone*, We insert a layer that records the time elapsed between the *Read* command is called and the *readDone* event is received. This is obviously an approximation of the time during which the ADC is actually turned on.
- For the CPU, we leverage the fact that TinyOS has a simple task scheduler that puts the CPU into sleep mode when the task queue is empty. The microprocessor is awoken via interrupts generated from internal or external peripherals. We record the time elapsed between the CPU enters sleep mode and the woke-up interrupt handler is executed as *idle* and the rest of the time as *active*.

In order to collect this trace, we (a) encode each state as a combination of bits (our mote vector is of dimension 8) we thus use 8 bits to encode the states, and we (b) output each bit of the state as an IO pin, using a second mote, that we call *LogRecorder*, that records the state transitions. This mechanism is very similar to the monitoring techniques devised for deployment-support networks[7].

[9] The source code of the benchmarks is publically available; we removed the reference due to double-blind reviewing

3.5 Data Acquisition Applications

We use simple data acquisition applications as workload for our experiments. We build them from building blocks: sample, compress, store, and send. We create 4 applications that increase the parallel behavior of these tasks from isolation to parallel sample and transmission:

SampleCompressStore is a simple state machine, that runs each step in isolation. As each sample is retrieved, it is then compressed, and once 10 samples are retrieved they are stored to flash. This cycle is repeated 9 times.

DataAcquisition extends the state machine from SampleCompressStore to retrieve the data from flash and transmit it. Again, each step in isolation.

SampleStoreForward is similar to DataAcquisition, except without the compression step.

DataAcquisitionAdv performs the same tasks as DataAcquisition, but interleaves the sample and transmit processes. Store is done in isolation.

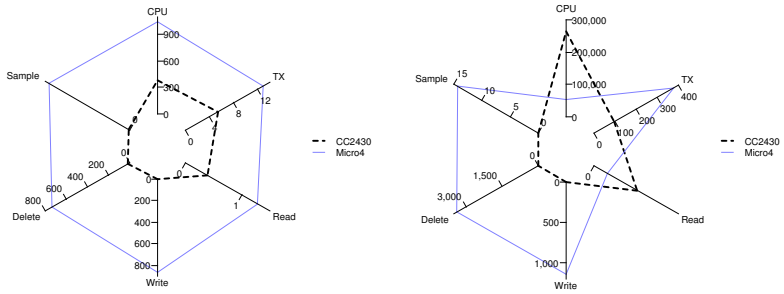
For our first experiments, we want a deterministic workload that exhibits reproducible results. One important source of variance in a sensor network applications is the environment. We choose a simple network topology and transmission scheme. Data is transmitted in 384 byte chunks (data and padding). The transmission does not expect acknowledgment that a packet is received, but only wait for the channel to be cleared (CCA) before sending. Sampling is at 10Hz and for compression we use the Lz77 algorithm.

4 Experimental Results

4.1 CC2430 and Micro

We ran the benchmarks described in the previous section on both the Micro and CC2430 motes. The time and energy mote vectors we obtain are shown in Figure 1 as spider charts. The results are somewhat surprising. CC2430 is much faster than the Micro when running the benchmarks. Slow memory accesses is compensated by the high clock rate. It means that the CC2430 can complete its tasks quickly, and thus be aggressively duty cycled. In terms of energy, we observe that:

1. CPU operations are two to three orders of magnitude more expensive on the CC2430 than on the Micro. This is due to the high clock rate (which guarantees fast execution) and to the overhead introduced by the slow access RAM.
2. Flash operations are much more expensive on the Micro than on the CC2430. These results led us to check our driver implementation (which is a positive results in itself). We could not find any bug. We believe that the difference in performance can be explained by the difference in clock rate between both platforms (1 MHz for the Micro vs. 32 MHz for the CC2430) and with the fact that the CC2430 driver is hand coded in assembler and the Micro's is not.

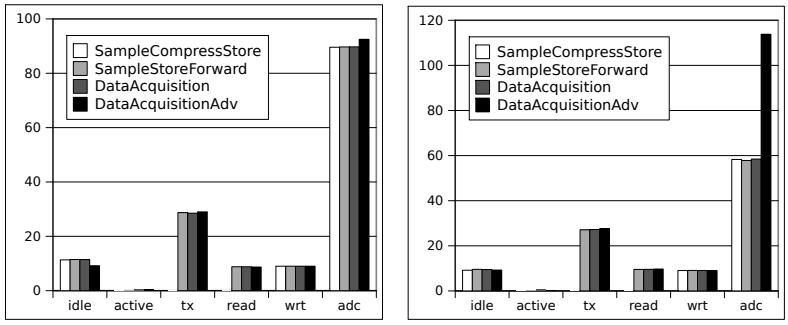


(a) Time Mote Vectors (b) Energy Mote Vectors

Fig. 1. Time and energy mote vectors for CC2430 and Micro

4.2 Performance Prediction

We used our methodology to derive the application vectors for the four data acquisition applications described in the previous Section. The results are shown in Figure 2.



(a) Micro (b) CC2430

Fig. 2. Application vectors for CC2430 and Micro

The profiles we get for the applications correspond to what we expect. Indeed, the application vector components for the ADC, flash and radio operations correspond roughly to the number of samples, flash and radio operations issued by the applications. The application vector is designed to be platform-independent. We thus expect that the application vectors derived from the CC2430 and Micro are similar. The good news is that they are at the exception of the ADC component.

This is either a measurement error, a software bug in the driver, or a hardware bug. We focused on this issue and observed that the time it takes to obtain a sample on CC2430 varies depending on the application. Two different programs collecting the same data through the same ADC driver experience different sampling times. We observed as much as 50% difference between two programs. We believe that this is another hardware approximation on the CC2430.

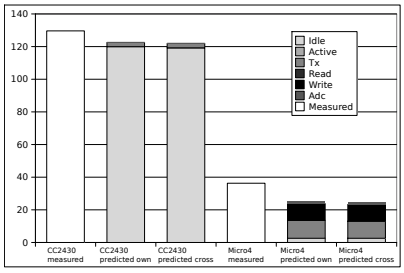


Fig. 3. Energy measurements and estimates

Our initial hypothesis is that the energy spent by an application on a mote can be estimated using the scalar product of the application vector with the mote vector. We computed the energy estimate for the *DataAcquisitionAdv* application and we compared them to the measurements we conducted directly on the motes (using an oscilloscope). The results are shown in Figure 3.

The estimations are well into an order of magnitude from the actual energy consumption. This is rather positive. As expected, the contribution from the CPU in active mode is insignificant. The poor performance of the CC2430 is due to the fact that we did not implement sleep mode support on the CC2430. Much more work is needed to test our methodology. This experiment, however, shows that we can use our method to prototype a data acquisition application with the Micro and predict how much energy the CC2430 would have used in the same conditions.

5 Related Work

The vector-based methodology proposed by Setlzer et al.[4] has been used to characterize the performance of web servers, OS utilities and Java Virtual Machines. Our paper is the first to propose this methodology in the context of sensor networks.

Performance estimation is of the essence for real-time embedded systems. The focus there is on timing analysis, not so much on energy consumption. We share a same goal of integrating performance estimation into system design [8].

In the context of sensor network, our work follows-up on the work of Jan Beutel that defined metrics for comparing motes[9]. Instead of using data sheets

for comparing mote performance, we propose to conduct application-specific benchmarks.

Our work is a first step towards defining a cost model for applications running on motes. Such cost models are needed in architectures such as Tenet [1] or SwissQM [2] where a gateway decides how much processing motes are responsible for. Defining such a cost model is future work.

6 Conclusion

We described a vector-based methodology to characterize the performance of an application running on a given mote. Our approach is based on the hypothesis that mote energy consumption can be expressed as the scalar product of two vectors: one that characterize the performance of the core mote primitives, and one that characterizes the way an application utilizes these primitives. Our experiments show that our methodology can be used for predicting the performance of data acquisition applications between Sensinode Micro and a mote based on the CC2430 System-on-a-Chip. Much more experimental work is needed to establish the limits of our approach. Future work includes the instrumentation of an application deployed in the field in the context of the Snowths project, and the development of a cost model that a gateway can use to decide on how much processing should be pushed to a mote.

References

1. O. Gwanali, KY. Jang, J. Paek, M. Vieira, R. Govindan, B.Greenstein, A. Joki, D. Estrin, E.Kohler. The Tenet Architecture for Tiered Sensor Networks. *Proc ACM Intl. Conference on Embedded Networked Sensor Systems (Sensys 2006).*
2. Rene Mueller, Gustavo Alonso, Donald Kossmann. SwissQM: Next Generation Data Processing in Sensor Networks. *Third Biennial Conference on Innovative Data Systems Research.*
3. Victor Shnayder, Mark Hempstead, Bor-rong Chen, and Matt Welsh. PowerTOSSIM: Efficient Power Simulation for TinyOS Applications. *Proc ACM Intl. Conference on Embedded Networked Sensor Systems (Sensys 2004).*
4. Margo Seltzer, David Krinsky, Keith Smith, Xiaolan Zhang. The Case for Application-Specific Benchmarking. *Workshop on Hot Topics in Operating Systems 1999.*
5. Vlado Handziski, Joseph Polastre, Jan-Hinrich Hauer, Cory Sharp, Adam Wolisz and David Culler. Flexible Hardware Abstraction for Wireless Sensor Networks. *Proc. 2nd European Workshop on Wireless Sensor Networks (EWSN'07).*
6. Reference removed for double-blind reviewing.
7. J. Beutel, M. Dyer, M. Ycel and L. Thiele. Development and Test with the Deployment-Support Network. *Proc. 4th European Conference on Wireless Sensor Networks (EWSN 2007).*
8. L. Thiele and E. Wandeler. Performance Analysis of Distributed Embedded Systems. In *The Embedded Systems Handbook.* CRC Press, 2004.
9. J. Beutel. Metrics for Sensor Network Platforms. *Proc. ACM Workshop on Real-World Wireless Sensor Networks (REALWSN'06).*