Abstract

Energy management is a critical concern in wireless sensor networks. Despite its importance, sensor network operating systems today provide minimal energy management support, requiring applications to explicitly manage system power states. To address this problem, we present ICEM, a device driver architecture that enables simple, energy efficient wireless sensor network applications. The key insight behind ICEM is that the most valuable information an application can give the OS for energy management is its concurrency. Using ICEM, a low-rate sensing application requires only a single line of energy management code and has an efficiency within 3.3% of a hand-tuned implementation. ICEM’s effectiveness questions the assumption that sensor network applications must be responsible for all complex management and cannot have a standardized OS with a simple API.

1. INTRODUCTION

Energy efficiency is a critical concern to mobile and battery powered systems. Reducing energy consumption improves system lifetime. An OS can improve energy efficiency by putting peripherals into low power modes and dropping the processor to a sleep state when idle. The challenge an OS faces is deciding when and how to do so: to manage energy well, an OS must infer future application behavior.

Prior work has shown us two things are generally true when an OS optimizes for energy: simple models are rarely effective and a bit of application knowledge can go a long way. For dynamic CPU voltage scaling, Vertigo [5] showed that having a process tell the OS its workload class greatly outperforms simple hardware heuristics and fixed-interval averaging [10], and GRACE-OS demonstrated that receiving explicit real-time deadlines from applications allows an OS to reduce energy further [33]. For disk spindown, Coop-I/O explored how application-specified timeouts on disk operations allow the OS to batch requests, outperforming even an oracle spindown policy for standard file interfaces [31].

Despite all of these advances, operating systems such as Linux, MacOS, and Windows use very simplistic energy management policies, such as preset processor speeds and fixed disk spindown times. The problem is that, beneath all of their advanced libraries, many applications today still use APIs which were designed before energy constraints were a major concern. The POSIX standard C library, for example, was standardized in 1989. This is the same year that NEC released the Ultrasite, the first notebook computer under five pounds. These APIs do not incorporate application information that enables an OS to make intelligent energy decisions. Since most dominant applications use these venerable APIs, including more advanced ones in an OS has limited benefit.

At first glance, wireless sensor networks (sensornets) seem to be a domain of computer systems that would avoid these pitfalls. Intended to last for months or years on small batteries, sensornets have harsh energy requirements, making energy management a critical component of almost every application. As sensornets have limited resources and very different use cases than traditional time-sharing systems, they tend to run small, customizable operating systems with flexible abstraction boundaries.

In practice today, however, sensornet operating systems, like their embedded siblings, provide minimal energy management support. They leave all complexity to the application. Embedded OSES such as eCos [24] and VxWorks [32], for example, have interfaces for processor power control but peripheral device control is up to an application. Sensor network OSES such as TinyOS [15], Contiki [4], MOS [1], and SOS [11] have power control interfaces which an application must explicitly invoke in order to change power states. Pushing all this logic into the application means that the OS does not prevent energy saving strategies, but this flexibility comes at the cost of application code complexity. For example, the core code for the TinyOS Great Duck Island deployment – the first successful long-term deployment of the OS – is 500 lines filled with special cases such as “if forwarding a packet, defer powering down.”

In this paper, we present ICEM (Integrated Concurrency and Energy Management), a sensornet device driver architecture that allows a sensornet OS to automatically minimize energy consumption without requiring additional explicit information from an application. The key insight behind ICEM is that the most important piece of information a sensornet application can provide is its concurrency. As most sensornet OSES are completely event-driven, an application simply needs to make all of its asynchronous system calls and let the OS schedule the underlying operations.

The research challenge in ICEM lies in the fact that some operations must occur serially. While application-level requests can use a single API call, peripherals often have complex call/return sequences, some of which have tight tim-
ing constraints. This requires integrating traditional blocking synchronization primitives into an execution model that is completely non-blocking. ICEM solves this problem by carefully exposing low-level concurrency to drivers through power locks, a novel non-blocking lock mechanism that integrates concurrency, energy, and hardware configuration management. For example, when a client acquires a driver’s power lock, ICEM has already powered and correctly configured the hardware for that client. When a power lock falls idle, ICEM powers down the underlying hardware. Power locks transform locks, a traditionally passive data structure, into an active energy management participant.

We have implemented ICEM by rewriting much of the TinyOS operating system, replacing its many ad-hoc policies and interfaces with three basic system abstractions and a small library of power lock components. We evaluate ICEM using a representative low duty cycle application that logs sensors every five minutes and sends data to a base station every twelve hours. Using ICEM, the application has a single line of power management code: it sets the radio duty cycle when it boots. For a representative monitoring application, ICEM achieves 96.7% of the energy efficiency of an implementation that optimally schedules I/O at the application level.

The rest of this paper is structured as follows. In Section 2, we introduce a sample application that we use as an example throughout the rest of the paper and provide background information on wireless sensor energy profiles and operating systems. In Section 3, we introduce how ICEM divides device drivers into three concurrency classes and give their defining characteristics. In Section 4, we describe how drivers manage their concurrency and energy, describe power locks, and provide details of the supporting power management library. In Section 5, we evaluate our approach using power traces of our sample application, runtime instrumentation of library calls, and a survey of where and how often the different driver classes appear. We compare our approach to prior work and existing sensor OSes in Section 6 and conclude in Section 7.

2. BACKGROUND

This section provides background on the challenges and details of managing energy on a wireless sensor. It starts with the sample application that we use as a running example through the rest of the paper. It describes the split-phase programming and concurrency model common to most sensor OSes. It overviews ultra-low power hardware characteristics and how they affect driver implementations as well as sensor-net energy management. From this information we observe that application concurrency gives a sensornet OS the ability to schedule operations in an energy efficient manner.

2.1 Application

Throughout this paper, we ground our goals and evaluation through a representative application of long lived, unattended wireless sensors. Every five minutes, the application samples four sensors and logs the readings in flash with a sequence number. Each log record is ten bytes. Every twelve hours, the application retrieves new readings from flash and sends them to a gateway. The application logs values to flash to provide data reliability in case of temporary or long-term disconnection, a common problem in long-term deployments [27].

Sampling and sending timings are completely decoupled: the two parts have a producer/consumer relationship on the log.

Data latencies of a day are acceptable for many low-rate environmental monitoring applications [26, 27]. If needed, the application can have a shorter reporting period, but an application saves energy by sending bursts rather than individual packets [19]. Low duty cycle sensornets generally keep their radio off using a technique called low-power listening [14]. In low-power listening, an idle node turns its radio on periodically, just long enough to detect if there is a carrier on the channel. If it detects a carrier, then it keeps the radio on long enough to detect a packet. The low power listening implementation used in this paper has a wake-up duration of 7ms: we configured the application to sample at 1Hz, giving the radio a 0.7% duty cycle when idle.

Because a low-power listener’s interval is typically much longer than a packet (e.g., one second), a transmitter must send its first packet many times in order to make sure the receiver has had a chance to hear it. The transmitter stops sending a packet once it receives a link-layer acknowledgment or it reaches a transmit timeout of twice the sampling interval. When a node receives a packet, it stays awake long enough to receive a second packet. Therefore, a packet burst amortizes the wakeup cost of the first packet over the follow-up packets. Figure 1 shows how the application’s transmission energy cost per packet decreases as its reporting interval increases.

2.2 Operating Systems

Packet bursts reduce how long the consumer part of the application stays awake when sending data to the base station. Concurrency reduces how long the producer part of the application stays awake when sampling. For each sample, the application needs to sample its four sensors and log the record...
Figure 2: Application pseudocode. Every operation is non-blocking. Rather than try to coordinate samples completion, the sampling loop starts a flash write of the previous cycle. Similarly, while it sends one set of samples it reads the next set from flash.

Figure 3: The application uses six peripherals: two SPI devices, two I2C sensors and two ADC sensors. The two I2C sensors are on the same chip (same I2C address), but require separate sensing command sequences.

to flash. Each of these operations is an I/O operation. Depending on the underlying hardware, the OS may be able to service some of them concurrently. To allow the OS to concurrently sample and log readings, the application keeps two sets of data buffers, one for the current period and one for the previous period. Figure 2 shows application pseudocode.

For an application to perform these five I/O operations concurrently, it must use non-blocking calls or allocate one thread per call. To provide concurrency with limited RAM, sensor network operating systems, such as Contiki [4], TinyOS [15], and SOS [11] generally take the first approach. In these OSes, every I/O and long-running system call is non-blocking and has a completion callback/upcall (there is no polling). For example TinyOS provides callback timers rather than a sleep call.

We use TinyOS because it supports the largest number of sensor node hardware platforms and therefore has the most diverse set of device drivers. TinyOS differs from other sensornet OSes in that it uses its own language, nesC, which is a dialect of C [6].

Sensornet OSes give applications explicit peripheral power control. The application must turn on a sensor before sampling it. Depending on the sensor, this can either be a split-phase (e.g., requires warm-up) or single-phase (ready immediately) call. For example, the telos platform – discussed in greater depth below – has two analog sensors. These sensors have a split-phase power interface because they depend on the microcontroller’s reference voltage, which takes approximately 17ms to warm up.

2.3 Hardware

We implemented our application in TinyOS on the telos platform. Telos has an 8MHz 16-bit CPU, a 250kbps 802.15.4 radio, a 2MB external flash, and 10KB of RAM [21]. The telos nodes we used had their external sensors populated. When fully active, a telos node draws approximately 30mA of current. In its deepest sleep state, it draws 37.6μA.

We defer detailed measurements of the telos’s power states and component latencies to Section 5 but at a high level the telos has two important considerations. First, on a pair of AA batteries, a node has an active lifetime of a few days. To last for months or years, a node must stretch this active time out by keeping its peripherals off and processor in a sleep state 99% of the time. Second, its microcontroller has a spectrum of low power states which keep different hardware resources powered. For example, a telos cannot drop to its 37.6μA sleep state when it is sampling the ADC or using a bus.

Figure 3 shows how the six peripherals the application uses are attached to the microcontroller. There are several opportunities for concurrency. Two peripherals are digital sensors that use the I2C bus, and two are analog sensors that share the on-chip ADC. The OS can concurrently sample one digital and one analog sensor. The OS must arbitrate resources between sensors of the same kind. The flash and radio are on the SPI bus, but the radio only needs the bus when loading a packet into transmit memory. Thus the OS can interleave flash reads and radio transmissions. Because the application schedules sensing and sending cycles independently, they can overlap and there can be up to seven outstanding I/O operations.

An OS can schedule and interleave application-level I/O calls, but soft real-time requirements mean some device drivers need to limit concurrency so they can perform small atomic sets of split-phase operations. For example, when the CC2420 radio receives a packet, its driver needs to perform two separate reads over the SPI bus. The first read is for the physical layer length field, which it uses to determine how long the second read must be. Other drivers require exclusive access to multiple resources at once in order to operate properly. For example, the telos platform, discussed in greater depth below, can configure its voltage reference to have different voltage levels; for some analog sensors, sampling requires not only exclusive access to the ADC but also to the voltage reference.

Microcontrollers minimize pin counts to reduce leakage current and packaging size. Therefore, MCUs often reuse pins for multiple purposes. For example, the MSP430 microcontroller has three bus protocols – I2C, SPI, and UART – that all share a common set of I/O pins. Only one can be active at any time, and the MSP430 has an intricate reset sequence for switching between them. The application’s digital sensors use a separate I2C bus. They do not share pins with the radio and flash chips SPI bus.

2.4 Managing Energy

Many peripherals have significant wake-up times, which waste energy by drawing current without doing useful work. To optimize for energy, an OS must minimize how often it
pays this cost. Scheduling I/O in bursts – whether it be ADC samples and powering up the reference voltage or sending packets and the need to wake up a receiver – amortizes this cost, reducing the average energy consumed per operation. Providing an OS the intended workload as a series of concurrent requests allows it use this concurrency to save energy.

But as the examples above showed, some low-level I/Os must be serialized and some device drivers require simultaneous access to multiple peripherals. In these cases, if a driver could tell the OS that it requires exclusive access to a resource, it provides both energy and concurrency management information.

This request is essentially a lock, but as TinyOS is completely non-blocking, it must be a split-phase lock. Furthermore, granting the lock to a device driver may first require configuring hardware. For example, granting the SPI bus to a telos driver requires reconfiguring the USART into SPI mode. However, on other platforms, such as the mica family [14], the SPI has a dedicated set of pins. Making device drivers platform-independent requires encapsulating hardware configuration into the lock abstraction itself.

3. ICEM DRIVERS

ICEM defines three broad concurrency classes: virtualized, dedicated, and shared. Figure 3 shows how the three differ in how they expose their concurrency to a user. This section presents the three classes in order of increasing complexity to the user. It gives concrete examples of when each class is used through the implementation of a stack for the radio used on most sensor nodes today. The next section describes how each class is implemented and how they work together to manage the overall power state of a sensor node.

3.1 Virtualized

Virtualized drivers are the simplest for a client to use: they have only a functional/data path. They support multiple users through implicit concurrency. Virtualized drivers buffer requests or otherwise maintain state in order to give each client the appearance of independence. It uses this per-client state to manage its concurrency, scheduling client requests in order to provide fairness or other desired properties. Because virtualized drivers can introduce latency, there is a tradeoff between programming simplicity and control. For this reason, application-level I/O interfaces are typically virtualized.

For example, a data-link packet sender is a virtualized driver. Each client can have a single outstanding packet. The packet layer buffers requests and services them with a round-robin policy, providing a limited (per-packet) form of fair queuing. Similarly, application-level millisecond timers are a virtualized driver. The driver implementation maintains per-client state, which it uses to schedule the underlying hardware timer in order to minimize the number of interrupts.

Maintaining per-client state allows a virtualized driver to automatically control its power state. For example, when a virtualized timer has no active timers, it disables interrupts and can stop the hardware counter if it does not require a real-time clock.

3.2 Dedicated

Dedicated drivers support a single user. Low-level hardware resources, such as counter/compare timer registers or a GPIO pin have dedicated drivers. Additionally, the lowest level of most hardware independent abstractions, such as packet sending, are typically dedicated. They give their single user complete control over requests and energy management. While they do not have software concurrency, dedicated drivers can handle hardware concurrency. For example, the USART on the MSP430 family of microcontrollers have a one byte FIFO for transmission, and so the dedicated USART abstraction can handle two pending requests.

In addition to a functional interface, dedicated drivers may provide an explicit power control interface. Not all dedicated drivers provide a power control interface, as sometimes power control is implicit in the functional interface. For example, enabling interrupts on a GPIO pin typically involves setting an enable bit in a control register: no additional on/off operations are needed.

3.3 Shared

Shared drivers provide explicit concurrency. They support multiple users, but users must contend for the driver through a lock. Shared drivers are typically abstractions which require or benefit from a single client being able to perform an atomic series of operations. Shared drivers are also used when an operation requires exclusive access to multiple underlying drivers at the same time. Calling commands on a shared driver without holding its lock is an error. Widely used shared drivers such as buses check that their lock is held by a caller, while narrowly used shared drivers such as an I/O pin driver assume those users are correct.

Like virtualized drivers, shared drivers buffer client requests: both allocate per-client state. The difference lies in what requests are buffered. Virtualized drivers buffer functional requests (e.g., send a packet), while shared drivers buffer lock requests. Since the lock abstraction is common across all shared drivers, its implementation can be a library that many drivers reuse. As we discuss in the next section, this library manages concurrency and pending requests it also manages a shared driver’s power state and configuration.

3.4 Example: CC2420 Stack

Figure 5 shows the component structure of the an example complex implementation, the ChipCon CC2420 radio stack. The CC2420 is the dominant radio used in sensornets today, due to its relatively high bandwidth (250kbps), range (up to 100 meters), and IEEE standard frame format (802.15.4).

At the top level, the CC2420 stack provides virtualized packet transmission and reception. When a client calls send(), the data-link layer virtualizer places the packet in its transmission queue. The virtualizer sits on top of a dedicated abstraction, serializing requests to its single functional
Figure 4: ICEM’s three driver classes: virtualized, shared, and dedicated. Dedicated drivers have a functional interface as well as an explicit power interface. Virtualized drivers provide implicit concurrency through a functional interface. Shared drivers provide explicit concurrency through a functional interface as well as a lock interface. Some shared drivers check client ownership, and some trust clients. Ovals are an instance of a client to a multi-client driver and a shaded box is an ICEM library component. Section 4.3 presents the details of the library.

Figure 5: The software structure of the CC2420 radio stack. It uses three SPI clients (shared), dedicated GPIO interrupts, one dedicated high-precision timer, one virtualized lower-precision timer, and provides a virtualized sending and receiving abstraction.

interface. As each sender has a single queue entry, the virtualizer gives each one a fair share of the packet transmissions. Protocols may and often do add additional queuing above. These virtualizations sit on top of dedicated abstractions of the send path, receive path, and control path. In addition to channel selection and transmit power, the control path implements the explicit power management interface of the dedicated abstraction.

The dedicated layers use several drivers, including a dedicated timer for MAC backoff and acknowledgments, a shared bus for interacting with the radio chip, dedicated IO pins, and a virtual timer for software IRQ on radio startup. The MAC timer is dedicated because it needs to be precise; the radio startup timer is virtualized because a bit of jitter when turning on the radio is acceptable. The bus is shared because multiple devices such as external flash chips and sensors use it, and because the radio must often perform atomic command/response sequences. For example, receiving a packet requires two separate reads. The first read is a single byte to get the length field of the physical layer frame. The value of the first read determines the length of the second read.

4. INTEGRATED MANAGEMENT

The goal of decomposing drivers into these three classes is to provide a limited set of concurrency and power management models that can represent most sensor network devices. The prior section described how these driver classes appear to a user and distinguished when each class is typically used. This section describes how they are implemented and how ICEM integrates distributed peripheral energy management with its centralized MCU sleep state control.

4.1 Driver Energy Management

As Section 3 described, dedicated drivers leave all concurrency and energy management to their client. Dedicated drivers provide explicit energy control through one of three interfaces: StdControl (single-phase control), SplitControl (split-phase control) or AsyncStdControl (single-phase control that is safe to call from within an interrupt handler). The lowest-level hardware abstractions are always dedicated, as they represent a specific physical resource, and they typically provide the AsyncStdControl interface so they can be called from code that needs immediate response to an interrupt. SplitControl and StdControl require going through the TinyOS scheduler, which can introduce scheduling latencies. Higher-level dedicated drivers, such as the radio, typically provide StdControl or SplitControl.

Virtualized and shared drivers integrate concurrency control and energy management. Virtualized driver implementations are interface-specific, as the interface they provide determines their required buffering. A virtualized driver therefore often has driver-specific power management logic, depending on what class of driver it sits on top of. The virtualizer is a central point of control: it is aware of all outstanding requests and the energy state of its underlying driver.

Shared drivers have a common interface to their concurrency mechanism: a lock. As synchronization primitives have difficult edge cases, designing shared drivers so they can share a lock implementation leads to simpler and more robust implementations. However, factoring concurrency out in this fashion complicates energy management, as concurrency state is no longer directly available to the driver. In practice, shared abstractions need to be on when a client holds their lock and can be off when no-one holds the lock. The lock itself therefore has the knowledge necessary to manage energy for the device. Rather than being a simple data structure or pushing complex state change callbacks into the core driver,
the lock protecting a shared driver becomes an active participant that manages the energy of the underlying hardware resources.

4.2 Split-phase Power Locks

ICEM supports shared drivers with power locks, synchronization primitives which couple energy and concurrency management. Traditionally, locks such as mutexes and semaphores are blocking constructs that protect critical sections or shared data structures. However, as TinyOS does not have blocking calls, its locks must be split-phase. A component calls a function to request a power lock and handles a callback when it acquires the lock. Systems whose timing is critical can try to avoid this split-phase operation when the lock is idle using an immediate request. Immediate requests, like operations such as pthread_mutex_trylock(), return if the lock was acquired but do not enter a waiting queue if not: the typical code pattern is to issue a standard request if the immediate request fails.

Power locks use recursive requests as a yielding mechanism. If a caller requests a lock when it already has a request pending, the request returns an error code. If a lock holder re-requests the lock, its request is enqueued: some time after releasing the lock it will receive another granted callback. Drivers that execute long or complex operations across several atomic actions often use a cooperative scheduling approach, re-requesting a lock as soon as it is granted and releasing it periodically.

Because TinyOS has a single stack, it has no threads. It therefore has no traditional OS concept of an execution context: there are no long-running sequential code sequences with serialized operations. Therefore, while there are locks which code can request and release, answering the question of who owns a lock is difficult. Our approach equates call sites with clients: each lock in the system can have a unique set of candidate holders. This set is determined by which components have registered with the lock interface. Unlike threads or objects, nesC components can only be created at compile-time. Therefore, a power lock’s candidate holder set is fixed at compile-time.

As power lock requests are non-blocking, a component can request several locks in parallel and proceed when they are all granted. This of course raises deadlock concerns. As not all power locks grant in their request order, it is possible for two drivers to request the same set of power locks in a specified lock order and still encounter deadlock. In practice, however, we have yet to find something like this occur. As power locks usually protect hardware resources, there is typically a clear series of lock requests encoded in the driver structure. Section 5.5 has one example of this, the light and temperature sensors of the MTS300 sensor board.

4.3 Component Library

Because power locks incorporate a queueing policy, a power management policy, and hardware configuration, ICEM provides a library of TinyOS components that allow driver implementers to easily combine small reusable building blocks into a complete power lock implementation. We distilled these components from our experiences reimplementing many of TinyOS drivers. The library has three types of components: arbiters, power managers, and configurators. Figure 6 shows these three classes of components and their interfaces. Clients principally use the Lock interface.

4.3.1 Arbiters

Arbiters are a power lock’s core component. Arbiters provide a split-phase lock interface that arbitrates between outstanding requests for a resource. A power lock’s arbiter determines its request queueing policy. In addition to standard clients that use the Lock interface, Arbiters can have at most one client which uses the DefaultOwner interface. A DefaultOwner cannot explicitly request the lock: instead, when the lock goes idle, the Arbiter defaults it to this special client. In effect, a default client always has an implicit lock request that is lower priority than any other client: the Arbiter automatically grants the lock to it when there are no other pending requests. Because the default client holds onto the lock whenever it is idle, the DefaultOwner interface has a callback to tell it when another client issues a request.

In addition to the Lock and DefaultOwner interfaces, Arbiters use the Configuration interface. Whenever the Arbiter grants the lock to one of its clients, it calls the corresponding Configuration command before signaling the granted callback to the client. If the client has not registered a Configuration implementation, it is a null call that nesC prunes away.

4.3.2 Power Managers

Power managers implement the DefaultOwner interface and use one of the explicit power control interfaces (StdControl, SplitControl, AsyncStdControl). A power manager specifies a power lock’s energy management policy. When a power manager receives the lock from its Arbiter, it takes this as a signal that the device is idle and may be powered down. When a power manager receives a callback that the lock has been requested, it takes this as a signal that the device needs to be used and must be powered up.
Because power locks are split-phase, devices that require split-phase power control do not pose any particular challenge. Consider a power manager that sits on top of a dedicated driver with a SplitControl interface. When it receives a callback that a client has requested the lock, it calls SplitControl.start(). Some time later, the operation completes with a SplitControl.startDone() callback. In this callback, the power manager releases the power lock as the underlying hardware is now ready for use.

### 4.3.3 Configurators

Configurators are part of a lock granting path. They implement the Configuration interface that Arbiters use. Configurators are for when power lock clients have a code preamble which should execute before they perform any operations. Rather than require clients to include this code in their lock grant callback, configurators allow them to incorporate it into the Arbiter’s logic. This is useful when many clients have the same code preamble: one implementation can serve all of them.

### 4.4 Sleep Energy Management

Power locks integrate concurrency and energy management for system peripherals. The second part of an efficient energy management strategy is to control the sleep state of a node’s microcontroller. While a node’s microcontroller typically has a low current draw compared to peripherals such as the radio, over long time periods the cost can add up. For example, a telos node can last approximately two months with all of its peripherals off but its microcontroller active. A telos node’s microcontroller. While a node’s microcontroller typically has a low current draw compared to peripherals such as the radio, over long time periods the cost can add up. For example, a telos node can last approximately two months with all of its peripherals off but its microcontroller active. A telos node’s microcontroller. While a node’s microcontroller typically has a low current draw compared to peripherals such as the radio, over long time periods the cost can add up. For example, a telos node can last approximately two months with all of its peripherals off but its microcontroller active. A telos node’s microcontroller. While a node’s microcontroller typically has a low current draw compared to peripherals such as the radio, over long time periods the cost can add up. For example, a telos node can last approximately two months with all of its peripherals off but its microcontroller active.
Table 1: Current draw, duration, and the lowest MCU power state of the major components on the telos platform.

<table>
<thead>
<tr>
<th>Device</th>
<th>Current</th>
<th>Time</th>
<th>Power State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microcontroller</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Active</td>
<td>1.92mA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>LPM1</td>
<td>182μA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>LPM3</td>
<td>3μA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Vref Off</td>
<td>+536μA</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>Radio</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Receive</td>
<td>18.86mA</td>
<td>NA</td>
<td>LPM3</td>
</tr>
<tr>
<td>Send (0dB)</td>
<td>18.96mA</td>
<td>12-2000ms (LPL)</td>
<td>LPM3</td>
</tr>
<tr>
<td>Flash</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read Record</td>
<td>1.75mA</td>
<td>5ms</td>
<td>LPM1</td>
</tr>
<tr>
<td>Write Record</td>
<td>2.69mA</td>
<td>5ms</td>
<td>LPM1</td>
</tr>
<tr>
<td>Analog Sensors</td>
<td>1.46mA</td>
<td>2ms</td>
<td>LPM1</td>
</tr>
<tr>
<td>Humidity Sensor</td>
<td>458μA</td>
<td>75ms</td>
<td>LPM3</td>
</tr>
<tr>
<td>Temperature Sensor</td>
<td>458μA</td>
<td>220ms</td>
<td>LPM3</td>
</tr>
</tbody>
</table>

Table 1: Current draw, duration, and the lowest MCU power state of the major components on the telos platform.

### 5.1 Microbenchmarks: Telos Energy

The instantaneous current draw of a telos node is a function of the microcontroller power state, whether the radio, flash and sensor peripherals are on, and what operations active peripherals are performing. Given a perfect measurement setup, the energy consumed by a node is the integral of a trace of its current consumption. However, the three orders of magnitude difference in a node’s power states make it unfeasible to measure at this precision. A measurement circuit that can accurately measure μA sleep currents cannot measure mA active currents, and one that measures mA currents is unable to measure μA currents due to circuit noise.

We therefore follow the methodology of earlier low-power studies [19] by measuring the average current draws of specific sleep states and I/O operations using a precision multimeter. To measure the time to completion of an operation we wrote applications that performed an operation once and observed the changes current draw on an oscilloscope. To measure the average power cost of I/O operations we wrote microbenchmarks that repeatedly perform one operation in an infinite loop. To measure the average cost of a sleep state we used the power-state override of ICEM’s sleep state calculation. To measure the length of each operation we connected a node to an oscilloscope and measured the timing of changes in current draw. To the best of our knowledge, these are the first published measurements for current (revision B) telos nodes: prior results are either derived from datasheets [21] or for the revision A, which has different components [20].

Table 1 shows the results of these measurements. The processor values are reported for each power state and include leakage currents of platform peripherals such as sensors, USB, and flash. The peripheral values do not include the processor current draw; instead, they show the lowest power state the processor can enter while that peripheral is in use. This is LPM3 for the radio because the SPI bus is off except for a few hundred microseconds of radio commands. It is LPM1 for the flash because logging operations keep the SPI bus on. It is also LPM1 for the analog sensors because the ADC requires a clock source, while it is LPM3 for the two I2C sensors because the bus is in software on GPIO pins. Finally, the voltage reference requires a 17ms warmup time after being turned on before it can be used.

### 5.2 Microbenchmarks: Power Lock Library

The measurements in Table 1 provide the basic costs of different I/O operations and sleep states. Using ICEM, an application can submit all of its I/O requests in parallel and let the OS schedule operations to minimize energy consumption. Compared to a hand-tuned implementation that optimally controls component from the application, ICEM adds overhead in three ways. First, the generalized library components use extra code space and RAM. Second, calls that sit on top of shared drivers must go through arbiters, which takes CPU cycles and therefore energy. Finally, ICEM drivers must be able to handle serialized requests, so must consider warmup times when deciding on power down policy.

The ICEM library has four arbiter implementations, supporting two arbiter policies and two levels of complexity. The two policies are round-robin (RR) and first-come-first-served (FCFS). The former scans across the clients in a deterministic order, while the latter grants the power lock to clients in the order they requested it. For each policy, there is a full arbiter implementation that supports power management and a simplified one that does not (it has no default client). In the figures and the rest of the text, a plus (+) denotes a complete arbiter (e.g., RR+) while a lack of a plus (e.g., FCFS) denotes a simple arbiter.

Table 8 shows the code size of the four arbiters as the number of clients increases.

Figure 8 shows the CPU cycle overhead of performing different arbiter operations. The most expensive is the com-
Table 2: Memory overhead of using the power manager library components

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>RAM (bytes)</th>
<th>ROM (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Immediate PM</td>
<td>Asyc 0</td>
<td>Std  5</td>
</tr>
<tr>
<td></td>
<td>Asyc 0</td>
<td>Std  0</td>
</tr>
<tr>
<td>Deferred PM</td>
<td>Asyc 4</td>
<td>Std  6</td>
</tr>
<tr>
<td></td>
<td>Asyc 4</td>
<td>Std  312</td>
</tr>
</tbody>
</table>

Figure 9: Cycle count overheads of common power lock operations. The telos processor runs at 4MHz.

The ICEM library currently includes two power management policies, immediate and deferred. Both turn on a resource as soon as they receive a request. They differ in when they turn it off. The immediate power manager turns off the device as soon as it re-acquires the power lock, while the deferred waits a period of time. On the telos platform, the bus and ADC have immediate power managers, while the voltage reference and flash chip have deferred power managers. Because the voltage reference has a 17ms wakeup time, our implementation gives it a 20ms power-down timeout. Therefore, if there are requests whose interval is less than 17ms, the ADC will be able to service them. If the request interval is 20ms or longer, then there is a 17ms jitter on some requests.

5.3 Application Performance

The telos power and arbiter cycle count measurements allow us to quantify the energy consumption of four different implementations of the low-rate sampling application. The first is our ICEM implementation following pseudocode in Figure 2. It uses decoupled sense and send timers and issues all I/O operations as a series of concurrent requests on top of ICEM. The second and third are single-threaded versions with serialized I/O calls so ICEM cannot use concurrency to improve energy usage. The two single-threaded versions differ in their sensor sampling order, which affects how often deferred drivers are powered on and off. The last version is an optimal implementation (pseudocode in Figure 11) in which there is no arbiter overhead and application-specific logic optimally schedules I/O operations and component power states.

ICEM’s effects are visible in the application-level sensing and sending logic. Figure 10 shows a detailed power trace of the ICEM implementation on the sense-and-log step, detailing the relevant events corresponding to the ICEM pseudocode. “Write prior samples” corresponds to the log write and log timeout events. “Sample photo active” and “sample total solar” correspond to vref warmup, analog samples, and vref timeout events. “Sample temperature” and “sample humidity” correspond to events of similar names as well as the digital sensor timeout, which exists in case the digital sensor does not respond. The digital sensors, which must be serialized, dominate execution time. Table 3 outlines optimal energy usage per day for the application logic (“Data Logging” and “Data Upload” sections), and Table 4 compares the four implementations.

Table 4 shows that ICEM executes the application logic efficiently. An ICEM node will have a lifetime that is 96.7% of the optimal implementation. In comparison to the optimal implementation, the ICEM version wastes energy in two ways. First, it spends time in active mode due to power lock overheads. Second, it has the timeout on the voltage reference, which the optimal implementation knows to turn off immediately after the second ADC reading. In the sensing cycle, there several arbiter operations, which all together are 1800 cycles per sample, for a total of 4100µAs per day. For sending, ICEM and the optimal implementations perform identically. The deferred power manager for the flash chip consumes a small amount of energy with its timeout timer, but as this occurs twice per day and is less than a millisecond long, it is negligible.

The single-threaded approach is significantly less efficient (79.1% at best); additionally, the order in which sensors are sampled has significant effects on its lifetime. This waste is predominantly because these implementations must ser-
Figure 10: Oscilloscope trace of one sample period for our example sensornet application, with spikes and important events labeled. Issues with our measurement circuit means that the measured MCU current levels are approximately 800µA above the accurate measurements in Table 1.

Table 3: Per-day energy consumption of the optimal implementation in µAs. The power numbers are different than Table 1 because they include the LPM state. A pair of AA batteries have approximately 2700mAh, or 9.72·10³µAs: a node with this duty cycle can last approximately 2 years. With no listening the node can last 8 years.

Table 4: Per-day energy consumption of the four implementations, considering first only application logic, then including sleep mode and low-power-listening costs. Serial 1 and 2 are the two single-threaded implementations, that differ in the order in which sensors are sampled.

5.4 Code Complexity

The major change from the ICEM to the optimal application is the addition of code to explicitly control the power state of all peripherals, as shown in Figure 11. Written in nesC code, the optimal implementation is approximately 400 nesC statements, with several edge cases due to the contentious for the SPI bus between log reads, log writes, and packet transmissions. In contrast, the ICEM implementation is 68 nesC statements, most of whose complexity is in correctly dealing with the seek pointer in the log. Of those state-
Every 5 minutes:
- Turn on SPI bus
- Turn on flash chip
- Turn on voltage reference
- Log prior readings
- Start humidity sample
- Wait 5ms for log
- Turn off flash chip
- Wait 12ms for vref
- Turn on ADC
- Start total solar sample
- Wait 2ms for total solar
- Start photo active sample
- Wait 2ms for photo active
- Turn off ADC
- Wait 34ms for humidity
- Start temperature sample
- Wait 220ms for temperature
- Turn off I2C bus

Every 12 hours:
- Turn on SPI bus
- Turn on radio
- Turn on flash chip
- Get next reading
- Wait 5ms for log
- Turn off SPI bus
- Wait for send

Figure 11: Pseudocode for the optimal application implementation.

<table>
<thead>
<tr>
<th>Driver Class</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dedicated</td>
<td>52</td>
</tr>
<tr>
<td>Shared</td>
<td>21</td>
</tr>
<tr>
<td>Virtualized</td>
<td>41</td>
</tr>
</tbody>
</table>

Table 5: Number of TinyOS components that fall into ICEM’s driver classes after our reimplementation.

ments, only one deals with energy management: it sets the radio’s LPL interval to be one second.

5.5 Example Drivers

For ICEM to be an effective architecture for sensor node device drivers, it must be able to handle the diverse peripherals that low-power sensors have. To test whether this is the case, we reimplemented the drivers for five TinyOS platforms (mica2, mica2dot, micaZ, telos, and eyes) and three sensor boards (mts100, mts300, telos). This represents a total of 114 drivers, with Table 5 showing the breakdown between the dedicated, shared and virtualized classes.

Next we present four example device drivers we have implemented. We chose these drivers because they highlight interesting requirements sensor network hardware can introduce and how our implementations use power locks to provide a simple API while meeting hardware requirements.

5.5.1 Atmega128 ADC

The atmega128 microcontroller has an on-board 10-bit analog-to-digital converter with 8 single-ended input channels. Configuring the atmega128 to take a sample requires configuring three parameters: input channel, reference voltage source, and clock prescaler. The ADC can use either an internal 2.5V source or an IO pin for a reference voltage. The clock prescaler allows software to adjust the ratio between the ADC clock rate and the core clock frequency in order to maintain timing requirements when the processor is running at a slower (and lower power) speed. Sensors typically only configure the channel and reference voltage.

Figure 12 shows the component structure of the atmega128 ADC driver. The ADC is a virtualized service that uses a round robin arbiter to service requests. To sample from the ADC, a program must instantiate an ADC client and provide a component that configures the ADC as needed. When a user of the client requests an ADC sample, the client requests the lock on the ADC from the arbiter and returns. When the lock is granted, the arbiter has automatically configured the ADC for a sample and the client requests a sample from the underlying ADC. When the ADC lock is idle, the arbiter powers down the ADC through the power manager, which clears the enable bit in the ADC control register. When the arbiter receives a lock request, it tells the power manager to turn on the ADC and the power manager sets the ADC enable bit.

5.5.2 MTS300 Photo Sensor

The MTS300 is a sensor board for mica-family nodes. It has a large number of analog sensors, including magnetometers, accelerometers, a microphone, a temperature sensor and a light sensor. The large number of sensors mean that two of the sensors – the light and temperature sensors – share an ADC input channel. Each sensor uses a separate GPIO pin to provide a source voltage: controlling which sensor is sampled requires having only one of these two pins active. The sensors require approximately 20ms to stabilize after voltage is applied. We describe the photo (light) sensor driver here;
the temperature sensor implementation is essentially identical except that it enables a different GPIO pin.

A photo sensor is a virtualized service that sits on top of the atmega128 ADC. A photo sample goes through three levels of arbitration. The first is the lock on the photo driver: only one photo client can sample at a time. The second is the lock on the shared ADC channel, which ensures only one of the photo and temperature drivers is active at any given time. Finally, sampling must acquire the lock on the ADC itself through an ADC client.

Figure 14 shows the component structure of the MTS300 photo driver. When a client requests a sample, the virtualizer requests the lock from the photo arbiter. The arbiter’s power policy turns on the photo driver with a split-phase call through a power manager. When turned on, the photo driver requests the lock on the ADC channel from the pin arbiter. When the pin arbiter grants the lock, the photo driver powers up the photo pin and starts a 20ms timer. When the timer fires, the driver indicates to the power manager that it is fully started, the photo arbiter grants the lock to the client, and the client requests a sample. This request goes through the ADC virtualization process described in Section 5.5.1.

5.5.3 MSP430 USART0

The MSP430 series microcontroller has two buses, USART0 and USART1. USART0 can be configured to act as an SPI, I2C, or UART. On sensor platforms such as Telos and eyes, USART0 is shared among a large number of peripherals. For example, on Telos nodes, USART0 is used as an SPI bus for the radio and external flash chip and as an I2C bus for external sensors.

Figure 14 shows the component structure of the USART stack. Unlike most other hardware resources, the MSP430 USART driver has multiple types of clients. Unlike the atmega128 ADC stack, which has client-defined configurations, the USART clients have predefined configuration settings. While some peripherals need to configure the buses in a special way (e.g., run at a slower speed), this is rare.

The USART clients are shared abstractions. Unlike the atmega128 ADC, which virtualizes ADC samples, many peripheral drivers require performing a series of uninterrupted requests where the bus is held between each operation. Therefore, the SPI, UART, and I2C clients export the lock interface, providing users access to the underlying lock on the bus. The configuration components mean that when a client acquires the lock, the bus has already been automatically configured for that protocol. Finally, if no client requires the bus, it is powered down and its interrupts disabled.

5.5.4 Storage

Figure 15 shows the component structure of the LogRead and BlockRead abstractions for the STM25P flash chip on the Telos platform. BlockRead provides atomic read/write operations on fixed-size and fixed-offset data units. Block storage is typically used for large data that needs random access, such as code or data downloads. LogRead provides atomic read/write operations on variable-sized records. Log writes are append-only but reads can begin at any record boundary. Log storage is typically used for storing streams of sensor readings.

Since NOR flash supports random access reads and writes, both log and block storage are implemented on top of a “sector” abstraction of the underlying chip. Because log and block present application-level units of atomic reads and writes, they are virtualized services. Each block or log client has an associated sector client that it uses to read and write the chip. As the block and log drivers need to perform multiple underlying reads and writes for an atomic operation, the sector interface is a shared abstraction. For example, when a block client requests to write a block, it tries to acquire the lock on the sector abstraction. When it acquires the lock, it performs the necessary operations to write a block atomically, releases the lock, and signals completion to the application.

The power manager wires to the sector virtualizer rather than the SPI client because power state changes on the STM25P require sending commands to it. When the sector arbiter determines that there are no active clients, it gives the lock to the power manager, which tells the sector virtualizer to shut down the chip. The sector virtualizer acquires the lock...
6. RELATED WORK

ICEM borrows heavily from three large areas of prior work: concurrency control, energy management, and I/O scheduling. Each of these topics has made key observations which ICEM incorporates.

6.1 Concurrency Control

Traditional concurrency control deals with threads and blocking synchronization primitives such as locks, semaphores, barriers, and monitors [16]. The fundamental assumption in much of this work is that the underlying OS is itself threaded. In contrast, the event-driven execution model typical to sensornet Oses has a single stack and requires split-phase synchronization primitives. For example, power locks grant a lock with a callback. Split-phase locks are not themselves new – the EARTH parallel model introduced them for use in SMP clusters [28] – but ICEM uses them in the low levels of an OS rather than in high performance parallel computing. In addition, TinyOS [15] contains several device-specific split-phase concurrency control interfaces, such as BusArbitration, which we took as a starting point in designing power locks.

As there is no inherent CPU parallelism, nonblocking optimistic concurrency control algorithms [13] have limited utility. ICEM’s shared drivers also differ from many traditional synchronization primitives in that they export low-level locks to higher layers, providing explicit concurrency control across system boundaries.

As power locks protect hardware resources rather than memory structures, they are generally not amenable to most traditional locking optimizations [22]. For example, given the cooperative scheduling typical of sensornet Oses, spin locks are useless. Similarly, as drivers rarely “read” hardware resources unless to modify their state, there are few many-reader situations which might make read/write locks useful.

6.2 Energy Management

As laptops have grown in popularity, so has interest in improving their lifetime through energy management. Prior work towards this end has, for the most part, focused on individual devices, such as disks [3] [12], memory [18], or the CPU [9] [5]. Venkatachalam et al. provide a very good survey of current and proposed approaches [29]. The basic challenge these systems face is inferring future application activity from prior requests. ICEM avoids this problem at low levels of the OS by promoting a nonblocking model and, where serialization is unavoidable, allowing drivers to specify their needs through power locks.

ECOSystem takes these approaches one step further, incorporating energy has a first-class resource in OS scheduling [34] [35]. In the ECOSystem model, applications have units of energy called “currentcy,” which they spend as they use the CPU or peripherals. As ECOSystem is intended for a multitasking system, one key benefit currency provides is a common resource to enable fair-share scheduling across multiple system devices. The ICEM and ECOSyst

6.3 I/O Scheduling

Over the past 50 years, operating systems have refined concurrency as a way to improve I/O performance. With many concurrent requests, disks can scan rather than seek [7], and network links can remain busy although each open socket has significant timeouts and latencies [30]. In some cases, concurrency management goes one step further, filling idle periods with prefetching to speculatively improve future performance [8]. ICEM also uses concurrency to reduce system latencies, but for the purpose of reducing energy consumption rather than increasing performance. However, reducing system latencies and response times decreases how long a node has to be on, thereby improving system longevity.

7. CONCLUSION

We present ICEM, a device driver architecture for fully event driven operating systems typical of ultra-low power sensornets. ICEM enables applications with no explicit energy management to operate within 3.3% of a hand-tuned optimal energy schedule. The key observation behind ICEM is that the most useful information an application can provide for an OS for performing power management is its concurrency. Using ICEM, application level concurrency allows the OS to save energy by efficiently scheduling I/O. This result suggests that the programming interface to a sensornet OS should allow an application to submit complex workloads for scheduling rather than provide a set of simple imperative commands.

8. REFERENCES


