Dynamic Resource Management in a Static Network Operating System

Kevin Klues[◦∓], Vlado Handziski[◦], David Culler[≀], David Gay[‡], Philip Levis[•], Chenyang Lu[∓], Adam Wolisz[◦]

[∓] Washington University, St. Louis, MO
[◦] Technische Universität Berlin, Berlin, Germany
[≀] Arch Rock Corporation, San Francisco, CA
[‡] Intel Research, Berkeley, CA
[•] Stanford University, Stanford, CA

Abstract

We present novel approaches to managing three key resources in an event-driven sensornet OS: memory, energy, and peripherals. We describe the factors that necessitate using these new approaches rather than existing ones. A combination of static allocation and compile-time virtualization isolates resources from one another, while dynamic management provides the flexibility and sharing needed to minimize worst-case overheads. We evaluate the effectiveness and efficiency of these management policies in comparison to those of TinyOS 1.x, SOS, MOS, and Contiki. We show that by making memory, energy, and peripherals first-class abstractions, an OS can quickly, efficiently, and accurately adjust itself to the lowest possible power state, enable high performance applications when active, prevent memory corruption with little RAM overhead, and be flexible enough to support a broad range of devices and uses.

1. INTRODUCTION

A sensor network application is a complex entity. A typical application receives, processes, routes, and sends packets, samples sensors, filters data, and logs data to non-volatile memory all within a very short amount of time. It runs multiple network protocols, such as collection, dissemination, aggregation and point-to-point, each having their own unique timing, bandwidth, and buffering requirements. Some applications interleave their operations coarsely, in order to minimize jitter between them. Others interleave them finely, in order to minimize response times.

All of this complexity must run using limited resources and even more limited energy. A typical sensor node consists of a simple 8-bit or 16-bit microcontroller, a low-power radio, a battery pack (typically AA or lithium-ion), non-volatile storage, and sensors/actuators. As limited as these resources are, energy budgets require that a node spend most of its time asleep, leaving resources idle 99% of the time. When a node is active, though, it must use what it does have quickly and efficiently, returning to sleep as quickly as possible.

Resource management therefore emerges as one of the principal challenges of sensornet application development. In particular, sharing memory and peripheral devices (the radio, timers, A/D converters, etc), and efficiently using the limited energy supply of a sensor network node ("mote") require complex interactions between many different parts of a system, each of which is often written by a different developer. Because of its energy cost and application importance, networking dominates the requirements placed on almost all resources in the system. More efficient memory use allows larger queues for more efficient bursts and fewer dropped packets; more efficient energy management allows the network to be active more often or last longer; efficient peripheral management enables safe and fast interleaving, reducing latency and awake periods through parallelism.

In this paper, we present a novel architecture for resource management in sensor network operating systems. The underlying principle in this architecture is to fix the set of resources and their clients *statically* (at compile-time), only allowing their states and current owners to be managed dynamically (at run time). For instance, our memory manager does not dynamically allocate objects, but allows a fixed set of cooperating clients to dynamically share a set of statically allocated objects. Taking this approach is quite different from traditional systems in which the number of resources and their clients can both change dynamically. The choice to break from this traditional approach was not arbitrary, and is fundamental to the constraints of low-power event-driven sensor network systems.

Two significant differences between motes and traditional real-time or desktop architectures drive the need for a new resource management architecture. First, sensor network applications are characterised by bursty event-driven execution, triggered by long- interval timers or packet reception. This differs greatly from the periodic workloads common to traditional real-time systems or the CPU-intensive workloads of desktop machines. Second, motes run on very different hardware. Their microcontrollers have multiple independent buses for peripheral communication, hide very little latency through caches, draw 1-3mA when active, draw 1-2μA when in deep sleep power state, and take microseconds to wake up. In contrast, low-power microprocessors intended for

embedded devices, such as the ARM Cortex M3 [3] or Intel PXA27X [13], are similar to desktop processors. They have a single system bus that all peripheral devices share, draw 10-100 times as much current, and take tens or hundreds of milliseconds to wake up.

These constraints and requirements have led to a fundamental rethinking of how to design operating systems. TinyOS, the first broadly used sensor network OS, deferred resolving the tension between efficiency and supporting diverse resource allocation requirements by taking an application-centric view. TinyOS provides only basic mechanisms for resource management and energy conservation, leaving it to application-specific code to institute the entire policy [8]. Every application has to build largely from scratch, and has to resolve all of its resource conflicts on its own. While pushing all of the resource management complexity to the application level may not get in the way, one purpose of an OS is to provide meaningful abstractions that make application programming simpler without a significant sacrifice.

Later sensor network OSes have dealt with the challenge of resource management by adopting long-established architectures, for whom resource management is a well understood problem. MOS, the Mantis OS, uses lightweight threads, relying on blocking semantics and pending requests [1]; SOS resembles a standard event-driven system, relying heavily on dynamic allocation and dynamically registering callbacks [7]; the Contiki OS straddles both domains, wrapping a pure event-driven execution in a thread- like abstraction (protothreads) [5]. Each of these OSes have a different philosophy and execution model underlying their abstractions. When a model does not have a ready answer to all the unique challenges sensornets pose, however, the answer is generally the same: leave it to the application.

The research contribution of this paper is its presentation and evaluation of novel resource management approaches for memory, energy and peripheral devices in a sensornet OS. These approaches combine static and dynamic techniques to achieve the robustness of static allocation and the flexibility of dynamic management. Memory management uses pools of fixed size objects without controlling their allocation point. The core scheduling loop manages the microcontroller power state by examining which subcomponents (timers, buses, etc.) are active while allowing subsystems to specify additional requirements as necessary. Peripheral management is based on split-phase lock system based on components called *arbiters* that provide flexible policies, support automatic power management, and reconfigure hardware when needed. We show that these abstractions are flexible enough to be reused in many places throughout a sensor network OS, simplify application development, and are efficient enough to not prohibit high performance applications. Furthermore, together they automatically minimize the power state of a node with little runtime overhead.

Figure 1: A mote from the Redwoods deployment.

Section 2 of this paper provides an overview of a recent sensor network deployment whose experiences motivate our resource management architecture. Section 3 describes the requirements for managing memory, energy and peripheral devices in sensor networks, including the approaches taken by current sensor network operating systems (Contiki, MOS, SOS and TinyOS 1.x). Section 4 presents the details of our abstractions for managing memory, energy, and peripheral devices, and section 5 gives an example of how to build a non-trivial system component using these abstractions. Section 6 evaluates the overhead associated with an implementation of these abstractions, and section 7 concludes.

2. A SENSOR NETWORK DEPLOYMENT

In previous work [16], a sensor network was deployed to monitor the microclimate of a redwood tree forest in Marin County, California. The deployment consisted of mica2dot motes, powered by a 3V, 1000mAh battery, and a suite of climate-monitoring sensors: temperature, humidity, and incident and radiated light (in both full spectrum and photosynthetically active wavelengths). They used an Atmel ATmega128 microcontroller with 128 kB of program flash and 4 kB of RAM running at 4 MHz, a 433 MHz radio from Chipcon (CC1000) operating at 40Kbps, and 512KB of flash memory. Figure 1 shows a photo of one of the deployed motes inside of its watertight packaging.

The motes were deployed in two redwood trees for a period of 6 weeks, and set to collect one reading from four of its sensors every 5 minutes. These sensor readings were logged to the data flash, and sent over a multihop mote network to a larger battery-and-solar-powered base station (a PC-class device) which stored received readings in a database and transmitted them over a GPRS modem. This deployment placed significant demands on

memory for storing and forwarding messages in the multihop network, on peripherals for taking sensor readings, logging data to flash and sending messages, and on energy for enabling the deployment to run for the entire 6 week period (the included battery was capable of powering the mote for approximately two days at full power).

The approach used to manage each of these resources was mostly ad-hoc. The mote software used in this deployment (TinyDB [8]) included its own memory allocator to dynamically allocate space for data collection queries, while networking queues were statically allocated using a fixed-size array. Ad-hoc code within each peripheral's drivers managed shared hardware resources when necessary. The application manually powered sensors on and off before each use. The radio had fixed duty cycle (switch on for 4s every sampling period), and time synchronization coordinated these wakeup periods. The microcontroller switched to a sleep mode based on examining its internal state, waking up mostly to handle timer and radio-related interrupts.

Revisiting this deployment in the light of our later experience, we can make a few interesting observations. First, the ad-hoc policies used to manage energy and peripherals (but not memory) in this deployment were the cause of several major problems. The deployment was 100 nodes, but only 33 collected useful data for a reasonable portion of the experiment. The remaining 67 died early in the experiment as they spent a large amount of energy unsucessfully trying to synchronize. The Also, during testing, we had problems using the full spectrum and photosynthetically-active-wavelength sensors shared microcontroller pins and this contention was the source of many bugs in development. The deployment avoided this problem by not using the full spectrum light sensor, which was unnecessary for the scientific goals.

Second, some of the ad-hoc approaches used here evolved into our current resource management abstractions: our current scheme for computing the microcontroller power state (Section 4.2) is similar to that in this deployment; our arbiters for resource management (Section 4.3) are influenced by ideas proposed in the arbitration scheme between the light sensors.

3. RESOURCES

To meet the challenges inherent to embedded sensor network design, a number of different operating systems have recently emerged that each take their own approach to resolving the conflict between energy efficiency, robustness, and ease of programming. Many of the features present in these systems have been developed using well known operating system techniques, while others are completely novel in design. Each system has been designed to provide a certain set of features, at the cost of sacrificing others. Fundamentally, however, each of these systems needs to provide provisions for efficiently managing memory, energy, and access to peripheral devices.

In this section, we present the requirements for managing these resources in a sensor network setting, and compare how the TinyOS, SOS, MantisOS, and Contiki operating systems manage each of these resources. In Section 4 we point out some of the problems that arise with using the approaches taken by each of these systems, and suggest ways in which they can be improved up. Our comparison is based on using MantisOS 0.95, SOS 1.7, Contiki 0.9.3 and TinyOS 1.15 with telosb [12] motes.

3.1 Design Considerations

By providing the proper abstractions, an operating system can greatly simplify application development. There is an inevitable tension between simplicity, generality, and efficiency: while file descriptors are a good enough abstraction for most applications, DBMSes manage disks themselves. One of the advantages sensornet OSes have today is that they are still very small. Users can reasonably easily replace almost any part of the system if it is insufficient for their needs. While allowing users to replace subsystems is valuable, the better the design, the less users need to.

Good system abstractions have three basic properties. First, they are *flexible* enough to be used in many different circumstances. The more flexible the abstractions, the fewer there need to be. Rather than force users to learn a large number of separate approaches, a few key principles can be used and reused many times. To continue to use the example of UNIX file, file descriptors are flexible enough to represent not only files but also sockets, pipes, devices, and system data (/proc). Second, good abstractions *simplify* application programming by providing needed functionality while hiding unnecessary complexities and details. For example, UNIX files allow applications to read and write data without having to worry about disk layout or data consistency. Finally, good abstractions are *efficient*. While simple and flexible abstractions are sometimes by themselves sufficient, these benefits are weighed against the cost in terms of lost performance.

3.2 Memory

Managing memory, refers to two separate concerns. The first is the allocation of physical memory into software-usable memory objects (allocation); the second is how memory objects are made available to and managed by software (management).

There are two basic approaches to allocating and managing memory. Traditionally, statically allocated memory has been managed statically, and dynamically allo-

cated memory has been managed dynamically. Static allocation refers to the process of setting aside certain blocks of memory at compile-time for use as a single object type during the duration of a running program. With static management of this memory, the ownership of each block is predetermined at compile time, and can never change. Dynamic allocation and management, on the other hand, allow memory blocks to be reused by many different components for storing different types of objects depending on the current demands of a running application.

Statically allocating and managing memory offers certain advantages over dynamic memory allocation and management. First, it provides a weak form of type checking, reducing memory access errors and ensuring more robust program execution. Second, it is simple and efficient. Statically managed memory never has to be allocated or freed during program execution, and will always be available for use when required. Dynamic memory allocation and management, on the other hand, can be much more flexible, at the cost of occasional access errors and the overhead associated with allocating and freeing memory. The size of a message queue, for example, can be allowed to grow and shrink dynamically depending on network traffic, thus freeing up memory for use by the rest of the system. The choice of which memory allocation and management scheme is most appropriate for a given situation depends on the level of flexibility, simplicity, and efficiency that is required.

TinyOS follows a purely static memory allocation and management scheme, while MantisOS, SOS, and Contiki all use some form of dynamic allocation and management somewhere. Every component in TinyOS allocates the memory it needs at compile time and passing pointers between these components is strongly discouraged. It is possible, however, for one of these components to statically allocate a block of memory and then choose to dynamically manage that block itself if desired (TinyAlloc as part of TinyDB). MantisOS allocates all of its memory statically, with the exception of threads, the networking stack, and a dynamically managed tree library. SOS uses dynamic memory heavily. A kernel is used to dynamically load and unload modules from the system, each of which require their own memory space to be dynamically allocated at load time. Each of these modules can then in turn dynamically allocate memory as needed. Contiki takes a middle ground approach. Its modules use static memory almost exclusively, but when loading a module, must dynamically allocate space for each of its modules variables.

The different approaches taken by each OS for managing memory can be problematic in some cases. Since TinyOS is designed to be a purely static operating system, creating a component that essentially works around this restriction is bound to cause problems. Memory access errors are much more frequent, and the system as a whole is less robust. For the rest of the OSs, most of their problems arise in the way their memory allocators have been implemented. In MantisOS, the standard compacting allocator it uses requires 6 bytes of overhead on each allocated memory block. While this overhead may not be too significant when blocks are allocated in large chunks (communication buffers - 64B and thread stacks 64-1024B), it can be overwhelming when they are allocated in smaller chunks (tree nodes - 8B). SOS has several different allocators, but the one used most often is a standard first-fit defragmenting allocator. Blocks of memory are allocated in chunks of 8-16 bytes, each containing a three byte header. As we will see in section 6 the way in which this memory allocator has been implemented is extremely inefficient and can result in long latency operations. The Contiki memory allocator is extremely streamlined and performs quite well under most circumstances. It is a best-fit-freelist allocator that allocated a new chunk of memory only when the freelist is empty checking if the new chunk will overlap with the stack. As with all dynamic memory allocation schemes, however, it does have some overhead in terms of performance and header sizes.

3.3 Energy

We base our definition of managing energy on a distinction between *chips*, hardware devices with a physical power state, and *peripheral devices*, logical units of functionality. A chip consists of one or more peripheral devices, each of which have their own power states that contribute to the overall power state of the chip itself. The microcontrollers used on motes typically have multiple peripheral devices (e.g., ADC, USART, Reference Voltage Generator) and external chips typically have one (e.g. Flash, Radio). Managing energy on a mote is the task of determining the lowest possible power state for all chips at any given time, given the power state of all peripherals.

In classical operating systems, peripheral devices can be fairly complex, with multiple power states and relatively complex APIs for power management. For example, the 500-page-long ACPI standard [2] specifies four different device power states: D0 – device active; D1, D2 – intermediary power states and D3 – device off. However, most peripherals used on motes have only two meaningful power consumption states: on and off. Consequently, external chip power management is generally straightforward: a request to switch the chip's peripheral device on or off simply switches the chip itself on or off.

For microcontrollers, there are two sets of power states to consider: the active power state – this typically depends on clock rate and supply voltage – and the sleep

power state – this typically depends on which of the microcontroller's peripheral devices are currently switched on or off. Current sensor network OSes only adjust the sleep power state, for two reasons. First, it is the most important, as motes spend most of their time asleep. Second, many current motes do not allow the active power state to be changed at runtime, unlike recent desktop and mobile processors.

TinyOS and MantisOS both provide standard interfaces to controlling the power to peripherals. TinyOS provides a reusable interface with `start` and `stop` commands, while MantisOS drivers that wants to be power managed, must implement a *dev_mode()* function that can be called to modify the power state of the underlying peripheral. Three distinct device power states are supported: on, off and idle.

SOS and Contiki do not provide any standard mechanisms for managing the power state of peripheral devices. Some peripherals implement on and off functions (e.g., the CC1000 on SOS); others do not (e.g., the CC2420 in Contiki, but not SOS).

Microcontroller power management, on the other hand is a bit more involved. Contiki and TinyOS are event-driven operating systems, so when the event/task queue is empty (and no interrupts are pending), the microcontroller can be put in to some sleep mode. Currently, Contiki always places the telosb's msp430f1611 [10] microcontroller in low-power-mode 1, where power consumption is approximately 75μA. This microcontroller has lower power modes using respectively 17, 2.0 and 0.2μA, with progressively more functionality disabled. In TinyOS, selection of an appropriate sleep mode (which must be enabled by the application programmer) is computed using a chip-specific function that typically examines its internal registers to determine which peripherals are being used at the current time. On the telosb mote platform this function chooses between the 2μA low-power-mode, the 75μA mode and no low-power operation based on the activities being performed by its USART, ADC and Timer peripherals.

The microcontroller power management in MantisOS is tightly coupled with the thread scheduling and supports two levels of power saving. When the scheduler ready queue is empty (because all threads are blocked on I/O, for example), the scheduler *implicitly* puts the microcontroller into an *idle* state that consumes less power than the active state, but still supports full peripheral functionality. For larger power savings, the scheduler needs *explicit* information from the threads in order to determine when it is safe for the microcontroller to go into a *sleep* state. The signaling is performed by a *mos_thread_sleep()* function (similar to the UNIX sleep() call) that threads can use to declare the intended duration of sleep. When all threads in the system are sleeping, the scheduler is free to put the microcontroller into a deeper power-saving state, with only a single timer left running to wake the threads up after the sleeping period is over.

When SOS puts the CPU into a low power state, it places it into the highest power state that allows all peripheral devices to operate correctly; in some cases (the OKI ARM microcontroller) this is a simply busy loop.

3.4 Peripheral Devices

Managing peripheral devices has two parts. The first is providing shared access to devices requiring dedicated use between multiple clients. The second is switching peripheral devices off whenever possible, consistent with the needs of the device's clients.

Access to these devices can be provided using either shared or virtualized services. A shared service gives clients full access to the peripheral at the cost of some form of access control. Virtualisation gives each client its own (possibly simplified) "copy" of the peripheral, at the cost of some runtime or latency overhead.

TinyOS includes one significant virtualized service, the timer. This service is implemented using ad-hoc code that does not follow a very precise structure or implement any specific set of well-defined interfaces. For non-shared services, the usual approach to resolving sharing conflicts is to have the command that requests an operation return an error code when the service is already busy. It is then left up to the client to retry the operation at a later time, based, e.g., on waiting for the service's completion event (i.e., waiting for the currently-being-processed command to complete). In the presence of more than two clients for a service, there is no way to guarantee fairness.

Only one platform in TinyOS attempts to provide any sort of peripheral device management. On the Telos platform, the bus shared between the radio and storage subsystems has mechanisms to prevent conflicts, but only once the system boots. Applications must manually interleave the initialization of these subsystems or one of them will inevitably fail. Beyond these two examples, TinyOS does not provide any peripheral management mechanisms.

SOS does not provide mechanisms for controlling access to shared resources. It requires that all peripheral device management be handled at the application level, without explicit support for interleaving operations of any type.

The driver architecture in MantisOS closely follows the POSIX model with all interaction between the users and the driver layer constrained to only four system calls: *dev_read()* and *dev_write()* for reading and writing data, *dev_ioctl()* for passing device specific configuration information and *dev_mode()* for explicit device power state control (Section 3.3).

For coordination of simultaneous access, a traditional mutual exclusion approach is used. Each driver maintains a simple "mutex" When the peripheral is locked for exclusive access, any other calling thread is queued in an associated waiting queue and blocked pending the release of the mutex lock by the current owner.

Peripherals are typically accessed in Contiki by calling a particular set of C functions to directly communicate with hardware (e.g., the telosb flash chip, serial port support). In some cases, these functions also communicate with a protothread (Contiki's lightweight, thread-like abstraction for event-based systems [6]) that implements part of the peripheral's functionality (e.g., the CC2420 radio). Events are signalled by peripherals either by calling a particular function from within an interrupt handler, or by signalling an event to a specific protothread.

There is no general-purpose support for implementing either shared or virtualized services (there is a prototype semaphore implementation, but it is not currently used anywhere). Some peripherals provide ad-hoc virtualisation (e.g., timers). Others deal with sharing through various mechanisms: providing only blocking functions (e.g., flash,[1] serial port), synchronization via global variables (e.g., the CC2420 radio and access to the I2C bus - these share resources on the telosb) and buffering (the TCP/IP networking).

4. DYNAMIC MANAGEMENT IN A STATIC OS

This section presents our novel abstractions for performing memory, energy and peripheral device management in a static sensornet OS. We have based the design of these abstractions on our experience with previous sensor network deployments and on the problems associated with the approaches taken by other sensor network operating systems. These designs have been made as part of the development of T2 [9], a successor to the original TinyOS operating system, and references are made to it where appropriate in order to clarify design decisions and give examples of how these abstractions can be used.

4.1 Memory

The static allocation approach, exemplified by TinyOS, greatly simplifies memory management. However, it is very limiting and can lead to inefficient allocation. Systems that require greater flexibility, such as the TinyDB database engine, end up implementing their own dynamic memory manager. Dynamic memory allocation has the benefits of more efficient use in lightly loaded systems and a graceful degradation model in heavily loaded systems. However, it also has all of the traditional drawbacks, such as out-of-bounds accesses and inadequate error checking. As these bugs effectively occur within kernel code (there is no user code), they can easily crash an entire node. The danger and difficulty of these kinds of failures have led the SOS operating system to introduce a software fault isolation system [15], which can introduce up to a 400% overhead on memory accesses; in contrast, a survey of the code base of the TinyOS operating system found a single memory access bug in tens of thousands of lines of code [14].

[1]This could be a problem, given that erasing a flash block can take 2s on the telosb's ST M25P flash chip.

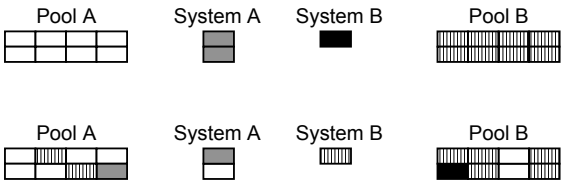

Figure 2: The memory pool abstraction. A pool allocates a static number n of objects at compile time (top figure) and dynamically manages n such objects at run time (bottom figure). Because every memory object in the system is statically allocated, the objects it holds may have been allocated elsewhere.

Our memory managers follow an approach intermediate between fully static and fully dynamic allocation: all objects are allocated statically, reside at a fixed address, and keep the same type. These objects, however, can be dynamically managed and shared between services and systems. As memory objects are of known size, overrun and underrun errors are rare. As objects keep the same type for the life of the program (even when "freed"), dangling pointers cannot cause memory errors (though may of course still cause application or service level bugs). This approach is similar to that of the SAFECode project. [4]

Figure 2 shows an example memory abstraction, the pool. A pool allocates N elements of a type t at compile time. Programs can allocate elements from and place elements into the pool. However, elements placed into the pool do not need to have orginated from within it. A pool nanages 0 to N elements of type t, and these elements could have been allocated anywhere in the address space. When the system boots, they happen to be the N elements the pool allocated. As the system executes, any memory object of the proper type can be placed into and allocated from the pool.

Buffer swapping is one way that objects allocated elsewhere make their way into the pool. Buffer swapping is a simple and commonly used technique to prevent memory starvation. When offered a memory object to use, a software module must return another object of the same type. This object may be the same one that was passed to it. Following this approach means that a component can-

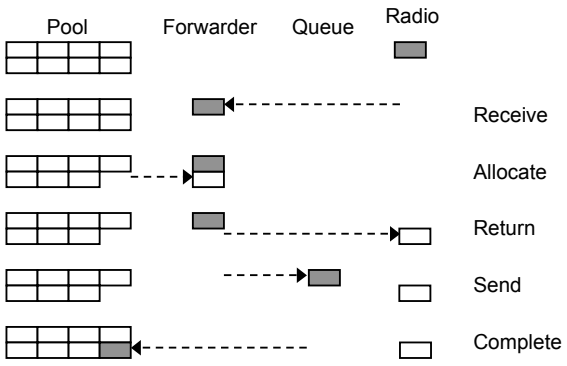

Figure 3: Buffer swapping example of how memory objects move in and out of a pool. A buffer originally allocated in the radio stack makes its way into the pool when a routing layer receives a packet to forward.

not acquire all of the memory in the system, as every exchange is tit-for-tat. Figure 3 shows this behavior. Consider, for example, a networking stack. When the stack receives a packet, it passes a packet memory object to a higher layer, which must return a packet to the stack so that the system can continue to receive packets. In order to return a packet, however, the higher layer might allocate one from a packet pool. As the stack passes packets to many higher layers, an object originally allocated by the pool may make its way into a completely different part of the system. Eventually, the higher layer frees the packet object it received from the stack into the pool.

4.2 Energy

Peripherals typically provide control over their power state by providing a standard interface which has `start` and `stop` commands. For peripherals on external chips, the implementation of these interfaces is straightforward.

Computing the power state of the microcontroller is significantly more complex. T2 puts the microcontroller into one of its low power sleep states whenever its task queue is empty and no interrupts are pending. The goal of T2's energy management system is to efficiently pick the best possible sleep state, based on the following information:

- The processor's hierarchy of power states. Each state typically has a set of onboard peripherals that are disabled, and an associated wakeup time (typically longer for lower-power states). We currently assume that power states can be ordered such that lower power states disable strictly more peripherals (as is typical), though our framework could easily be extended to handle more complex scenarios.
- The currenly active on-chip peripherals. The state of these peripherals is generally checkable by inspecting an MCU's many control registers.
- Any application or system requirements on power state and wakeup time. For instance, if a timer is scheduled to happen in 1ms, then a sleep state with a 10ms wakeup time is inappropriate.

These goals are achieved in T2 by a combination of three mechanisms:

- A platform-specific function (mcuPowerState) that computes the best sleep state based on which peripherals are enabled.
- An (optional) dirtyMcuState bit which must be set by any code that affects state checked by mcuPowerState. This avoids having to recompute mcuPowerState every time sleep mode is entered.
- Hooks by other parts of the system or application requesting (at least) a particular sleep state. These are necessary to handle any peripheral state not checkable (or checked, see below) by the mcuPowerState function, e.g., time-varying state like the time to the next timer event.

In C-like pseudocode, T2's sleep logic thus looks like:

```
sleep_state_t currentSleepState;
bool dirtyMcuState;

void enterSleep(void) {
  disable_interrupts();
  if (dirtyMcuState) {
    dirtyMcuState = FALSE;
    currentSleepState = mcuPowerState();
  }
  setSleepState(MAX(currentSleepState, sleepHook()));
  atomically_enable_interrupts_and_sleep();
}
```

Much of the details of this sleep logic are necessarily platform (rather than just MCU) specific. For instance, the mica family motes choose to enter "power save" mode, which disables some hardware clocks, even when those clocks are running. In effect, that means that those clocks are only measuring non-sleep time. Without this choice, these motes would never enter low-power sleep modes. Mica family motes also contain a hook that checks the delay until the next timer interrupt. If this delay is below 12ms, the mote enters "extended standby" rather than "power save". "Extended standby" wakes up from sleep in 6 cycles rather than 65536 for "power save". However, this distinction is only necessary because mica motes use an external crystal; when using an internal oscillator, wakeup from "power save" is also 6 cycles.

As a result of these and other similar considerations, T2 has a standard MCU power management architecture, but the implementations are necessarily heavily platform-specific.

4.3 Peripherals

In T2, peripherals are accessed via either virtualised or shared services. With the exception of timers, all virtualised services are built upon an underlying shared service with access controlled by *arbiters*, a lock-like abstraction. The timer implementation is ad-hoc, so we concentrate here on how arbiters support efficient sharing and power management of peripherals.

Access to each shared service is controlled by an arbiter, which offers a number of standard interfaces to both the service and its clients. These interfaces are: `Resource` which allows clients to request access to the service, `ResourceRequested` which informs a client that another client is interested in the resource, `ArbiterInfo` that allows a service to check that a client "owns" the arbiter, and `ResourceConfigure` which simplifies management of per-client service configuration. Additionally, arbiters offer a `ResourceController` interface which allow for automatic peripheral power management; this is discussed further below (Section 4.3.1).

`Resource` is the basic interface to an arbiter. An arbiter has `request`, `immediateRequest` and `release` commands to respectively request and release access to a service. With `immediateRequest` access is granted immediately if possible and denied otherwise, while `request` puts the client on a queue and grants requests in some arbiter-specific order.

Arbiters in T2 combine static and dynamic approaches. The set of clients of a service, and hence of its arbiter, is fixed at compile-time. This allows the arbiter to reserve space proportional to the number of clients, and hence ensure that its queue never overflows (however, each client is only allowed one outstanding request at any time; this can also be viewed as n 1-deep queues). Our experience shows that it is much easier to write reliable code when one is guaranteed exactly one queue slot, versus an unbounded put possible zero number of queue slots. Finally, note that ensuring that clients do not fill up the queue by making multiple requests does not prevent a client from monopolizing a service by not releasing it in a timely manner.

The decison about whether to use `request` or `immediateRequest` depends on the client requirements. If a client is only interested in obtaining a resource "right now", due to timing constraints or the semantics of its protocol, then it should use `immediateRequest`. Additionally, a client can choose to use `immediateRequest` to reduce latency, at the cost of a little extra code complexity to handle immediate request failure (typically just a call to `request`).

Sometimes it is useful for a client to be able to hold onto a resource until someone else needs it and only at that time decide to release it. The arbiter makes this information available via the `ResourceRequested` interface, which signals an event to the client on every request.

A misbehaving client may fail to wait to be granted access to a service before trying to using it. Service implementations may include runtime checks to detect this, by using the `ArbiterInfo` interface. This allows a service to check if an arbiter is currently granted to a client, and compare that client's identity to the one attempting to use the service.

The `ResourceConfigure` interface is used by an arbiter to automatically configure a resource for use by its client before granting access to it. It is also used to unconfigure a resource just after it has been released. A client specifies which hardware configuration the `ResourceConfigure` interface should be associated with and the arbiter automatically handles the configuration.

4.3.1 Peripheral Power Management

While the application programmer can simply explicitly power non-shared peripherals on or off, a safe control of the power state of shared peripherals is almost impossible without OS support, given the large number of ongoing activities in highly dynamic sensor network applications.

Dynamic power management of shared peripherals that have only two power states effectively requires tracking their ownership[2] in order to detect when the peripheral is unused, and thus can be powered down.

In some traditional OSs, this information is centrally available in the kernel's "device queues" that list the waiting processes blocked on I/O requests to/from a particular shared peripheral. In T2, this information is tracked in a distributed fashion by each peripheral's arbiter. Thus, instead of going with a centralized solution where a system-level power management component *observes* peripheral status and issues power-state transition commands, we continue the decentralized approach by introducing reusable *PowerManager* components that are tightly coupled with the *arbiter* in order to provide local and customizable power management.

These power management components connect to the peripheral's arbiter as illustrated in Figure 4. When the peripheral is not being used by any of the normal clients, the arbiter transfers the ownership of the peripheral to the power manager via the `ResourceController` interface. The power manager applies an appropriate power management policy and automatically powers the peripheral on or off via its `start` and `stop` commands that are part of the `StdControl` or SplitControl inter-

[2] As opposed to tracking full workload information for peripherals with multiple power/performance states.

faces.

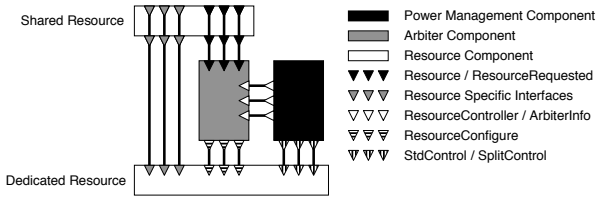

Figure 4: Component graph showing relationship between a resource, an arbiter, and its power manager

The optimal power management policy depends on the application requirements as well as on the time and energy overhead for the power state transitions. Thanks to the decentralized approach, the policy can be customized on a peripheral by peripheral basis, allowing better control of the trade-offs. Currently, T2 has reference implementations for two power management policies: a simple immediate policy and a deferred policy with fixed timeouts.

Under the simple policy, the power manager immediately powers the peripheral down after receiving ownership over the peripheral. The manager remains in ownership of the peripheral while it is powered down. Upon receiving information from the arbiter that some client wants to use the peripheral via the `requested` event of the `ResourceController` interface, the power manager powers up the peripheral before releasing it to the arbiter so that it can be assigned to the requesting client. This policy is optimal for the cases when the power state transitions carry negligible delay and energy penalties.

When the state transitions carry significant delay and energy overhead, the simple policy is no longer effective and can cause more harm then good. To prevent frequent state transitions, a simple mitigating technique is the introduction of an idle state prediction timer that is started when the power manager becomes the owner of the peripheral. In contrast to the simple policy, the power manager will not force a transition to the *off* state until the expiration of the timer. If a client requests the peripheral while the timer is active, the power manager cancels it and immediately releases the peripheral.

5. NETWORK STACK EXAMPLE

To demonstrate the simplicity and flexibility of our abstractions, we provide an example of how they can be used to build a simple networking stack. This networking stack consists of a radio driver (using many smaller subcomponents), a routing component, and a message pool.

The radio driver statically allocates a single message buffer for receiving incoming packets from the radio hardware and a pointer to a second message buffer for sending outgoing packets passed to it by the routing component. The routing component statically allocates a fixed sized queue of message buffers for storing packets that need to be forwarded, and an array of pointers to messages contained in the message pool for swapping with higher level components trying to send packets. [3]. Messages are then exchanged between the radio, the routing component and any upper level components using the buffer swapping mechanism described in section 4.1.

Along side all of the dynamics occurring at these higher layers, the low level radio driver internally implements a Low-Power-Listening (LPL [11]) sleep scheduling protocol integrated with a CSMA based MAC. This sleep scheduling protocol periodically wakes up the radio hardware in order to check if there is any activity on the radio channel. If there is, it leaves the radio on in order to receive a packet. If there is not, it turns the radio off and waits until the next check period. Whenever the radio has a packet to send it turns the radio on, checks to see if the channel is busy and sends if it is not. If it is, it continually waits some random amount of time (leaving the radio on in order to receive whatever packet is currently being sent on the channel) and then tries again until it is successful. All packets are sent with a preamble length equal to the check period in order to ensure that other nodes will wake up before the packet itself is actually sent.

Implementing this protocol can quickly become complicated without the help of proper OS support. The radio channel must be checked for activity by taking an RSSI reading using an ADC. A timer must be used to run the sleep schedule and random backoff timers for the radio. Communication with the radio must be performed over a USART component running an SPI protocol. By virtualizing access to each of these services using the techniques described in section 4.3, the implementation of the radio logic can be greatly simplified. The radio can act as a client to the ADC, Timer, and SPI services, only requiring each of their underlying resources to be active when in use. All the radio has to do is express its wishes to use one of these services, and the underlying software takes care of the rest. Requests are put into arbiters, resources are configured for proper use, and the microcontroller is put into its lowest possible power state whenever everything has gone idle. Furthermore, whenever the radio has been put to sleep using LPL, its hold on the SPI service can be released and the underlying USART can be automatically powered off using one of the peripheral device power managers. Without this support, it would be very difficult for the radio to ensure that it had

[3] A single node may be running multiple networking stacks and the message pool size must be large enough to accommodate all of them

exclusive access to each of these components and that each of them was powered down whenever they weren't being used.

6. EVALUATION

This section evaluates the proposed resource management mechanisms outlined in Section 4. When appropriate, we compare the mechanisms we have implemented in T2 against the approaches taken in other existing sensornet OSes like MOS, SOS and Contiki.

6.1 Memory Management

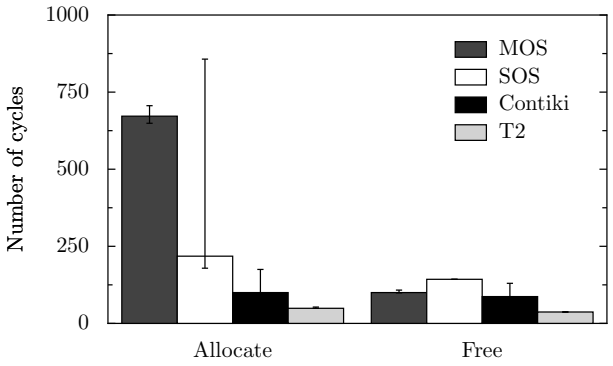

Figure 5: Cycle counts for allocating and freeing packet buffers for the memory management systems of sensornet operating systems. As TinyOS has no dynamic allocation, it is not present.

We evaluate the *efficiency* of different memory management approaches by considering a simple common use case in sensornets, a packet queue. For each approach, we simulated a networking component with a queue of up to 16 packets. Every 50ms, the component randomly either allocates or frees a packet. If the queue is empty, it always allocates, if it is full it always frees. This causes the queue depth to be a random walk. We let this program run for several hundred iterations. Packet buffers were 44 bytes. Figure 5 shows the results.

6.2 Energy Management

Measuring the energy consumption on real mote hardware is the only way to positively validate the proper functioning of the energy management policies of a sensor node OS. Because many of the management actions (especially the ones controlling the MCU low-power state) operate on fine time scales, the best insight can be gained by capturing detailed current consumption profiles at fine resolutions.

In this subsection we present the results from a battery of such current consumption tracing experiments that demonstrate the proper operation of the energy management policies presented in section 4.3 and implemented in T2. For reference, we have performed a corresponding set of experiments on MOS also. We have selected MOS because it is the only other sensor node OS that has comparable energy management feature set (summarized in section 4.2) as T2, and because it allows us to illustrate how the core OS assumptions (i.e. event-based vs. thread-based approach) can influence the flexibility and the performance of the energy management tasks.

The traces were collected using a standard *low-side* current measurement setup, with the potential drop over the small sense resistor amplified using an amplifier circuit before being sampled using a high-speed digitizer. Because we wanted to test both the MCU and the peripheral energy management capabilities, we have selected a telosB node populated with all on-board sensors as our system under test.

6.2.1 *MCU power state*

Our first experiment (figure 6 shows the current consumption levels of the standard T2 `Null` application that does little more than basic boot time initialization of the mote. An existing policy in T2 requires that the 32 kHz timer remains always active, effectively making the "Low Power Mode 3 (LPM3)" state of the MSP430F1611 the lowest power-saving state safe on telosB. The default curve shows that the `McuPowerSleep` function properly selects the optimal sleeping state in this simple scenario without tasks. For the other two curves (LPM1 and Active), we have used the `McuPowerOverride` interface to demonstrate how the application can override the default decision of the MCU manager. The figure shows that when the MCU is fully active, the mote consumes about 50 times more energy than when the MCU is in the default power-saving LPM3 state.

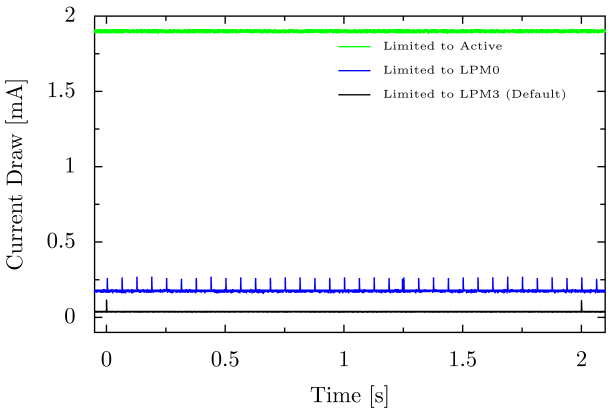

Figure 6: Current consumption of the `Null` application in T2 under different application MCU power state overrides

In our the second experiment (figure 7), we have

recorded the consumption profile of a simple T2 application that periodically posts two tasks. The first task is re-posted every 250 ms and as workload busywaits on a hardware clock for 50 ms. The second task is re-posted every 100 ms and busywaits for 100 ms. The durations and the periods were selected such that execution of the two tasks lines up each 500 ms, interleaved by a single execution of the shorter task.

The trace shows that even with multiple tasks active in the system, the automatic power management in T2 safely puts the MCU in the optimal power saving state whenever the task queue of the scheduler is empty and properly wakes it up when a task is posted for execution. Thanks to the superb wake-up capabilities of the MSP430F1611 microcontroller, the transitions are almost instantaneous (only a few us).

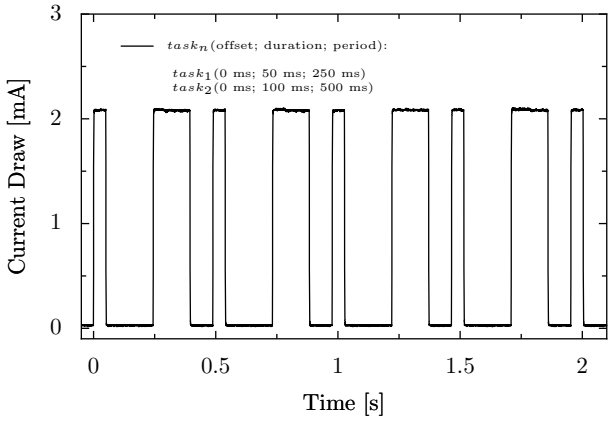

Figure 7: Current consumption profile of a simple T2 application with two periodically posted tasks

To compare the operation of the MCU power state management in T2 with the one in MOS, we have implemented the equivalent MOS application using two threads and recorded its profile (figure 8). The first thread busywaits for 50 ms using `mos_mdelay()` and then calls `mos_thread_sleep()` asking to sleep for 250 ms in an endless loop. The second thread does the same, but with busywait period of 100 ms and sleep time of 500 ms.

The obtained results show several interesting differences. First, the current consumption in the sleeping state is significantly higher even from the active state in the T2 run. Looking at the MOS source code, it seems that all pins and peripherals on the telosb mote are not properly initialized into their lowest power consuming states during the boot-up as in T2, results in significant base offset. While the MOS scheduler properly switches the MCU power state when both threads are in the sleep state, the relative change of 1 mA is lost in the base consumption of 3.5 mA. Another interesting observation is that the higher consumption period associated with the time when both threads are active is wider than the 100 ms duration of $thread_2$ due to the context switching overhead. As the trace shows, this time penalty is directly translated into increased power consumption because the MCU can not be placed in the lower consuming power state as early as in the T2 case.

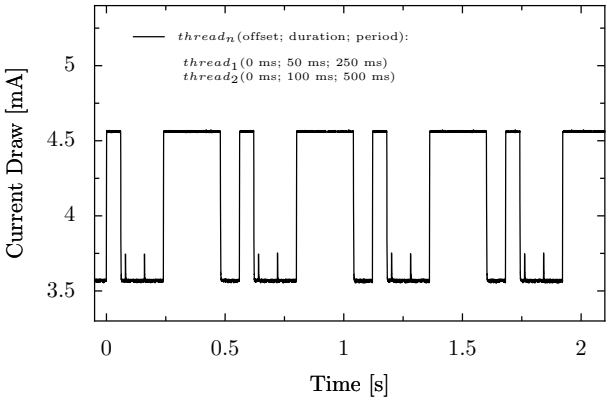

Figure 8: Current consumption profile of a simple MOS application with two periodically sleeping threads (MOS 0.9.5)

6.2.2 *Peripheral power state*

The base offset in the previous test shows just how important proper management of peripherals can be. In almost all existing sensor node platforms, the peripheral devices, especially the radio and the non-volatile storage, consume several times more energy than the MCU when active. Thus, proper management of their power states should be a primary task of any sensor node OS.

To illustrate the relative impact on the power budget by the different peripherals in a typical sensor node application, we have collected the current consumption traces of our telosB node running a modified `Oscilloscope` application in T2. Driven by a timer, this standard T2 application collects ADC samples from a sensor and stores them in a buffer. When enough samples are collected (by default 10) the buffer is sent over the radio, to be collected by a `BaseStation` application for visualization on a host computer. To demonstrate the arbitration support in T2 and its impact on the power consumption profile, we have modified the basic application to not only send the buffer over the radio, but to also store it into the flash. Since both the radio and the flash on the telosB platform are using the same SPI bus to connect to the MCU, this task requires proper arbitration of this shared resource.

Figure 9 shows the current consumption profile of the application when compiled without peripheral power management of the radio. The trace vividly shows how

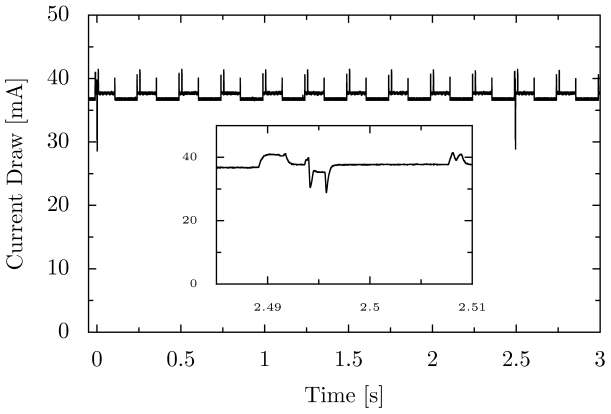

Figure 9: Current consumption profile of the original `Oscilloscope` T2 application without peripheral power management

the complete budget is dominated by the large consumption of the radio chip in receive mode. The increase in the consumption triggered by the MCU waking up on the timer interrupts, and the subsequent ADC sampling is visible as periodic spiking on top of the radio baseline. Every 2.5 s (i.e. after every ten buffered ADC samples), the radio switches into transmit mode and sends the buffer. Because the CC2420 chip consumes less energy while sending than while receiving, this shows up on the trace as periodic dips of about 10 mA. The inset shows this time window with finer resolution. The bulge at the start of the window is a result of the microcontroller waking up to service the timer event, the ADC sampling the sensor and the SPI bus arbiter reconfiguring the USART device using the `ResourceConfigure` interface. The flat line that follows covers the period while the message is being transferred over the SPI bus to the radio buffer. After the radio transits into transmit mode and sends the packet, the consumption is returned to the old level before the next timer interrupt and ADC sampling occurs.

The profile looks completely different when the power state of the peripherals in the system is properly managed (figure 10). Due to the periodic nature of the application, the radio in the modified `Oscilloscope` application can be explicitly controlled by the application code, while all the other shared peripherals, including the flash chip, are implicitly managed by the corresponding `PowerManager` components attached to their arbiters. This leaves the system into a very low power consuming state that is only disturbed every 2.5 s. The inset zooms on one such active phase. In contrast to figure 9, the peek consummation in the active phase is almost 10 mA higher because of the simultaneous activity of the radio and flash chips. But this is more then compensated by the low consumption in between the activity picks, since the energy consumption of the system is proportional with the area below the trace curve.

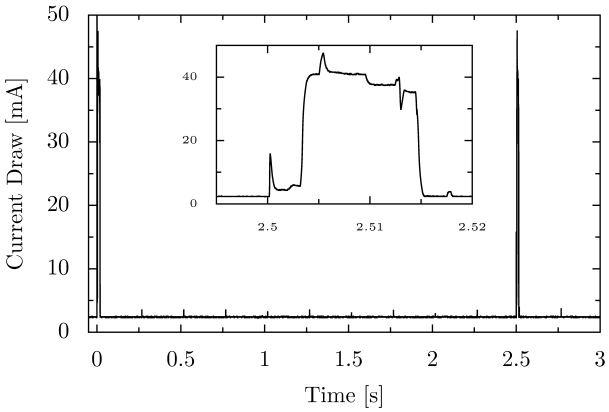

Figure 10: Current consumption profile of a modified `Oscilloscope` T2 application with explicit power management of the radio and implicit power management of the flash chip

6.3 Peripheral Management

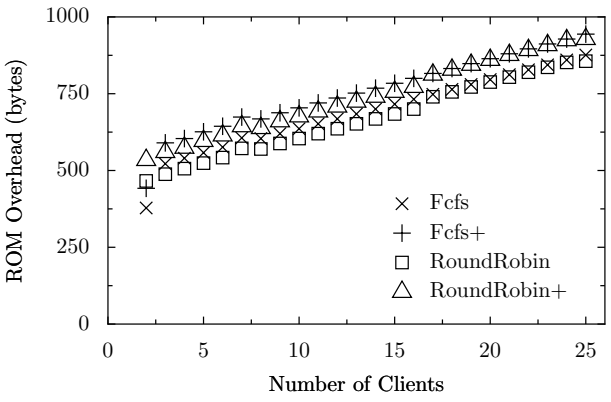

Figure 11: Program memory overhead using an arbiter.

Default arbiters have been implemented in T2 in order to promote code reuse between different resources requiring the same type of arbitration policy. Because of the semantics associated with the use of the `ResourceController` interface, arbiters that implement it tend to have a much larger overhead (in terms of both memory and execution cycles) than those that do not. Since this interface is only intended for use by shared resources that require a default user (e.g. a PowerManager), only some of the default arbiters implemented in T2 provide this interface. Four arbiters have been provided in total, two implementing a first-come-first serve (FCFS) arbitration policy, and two implementing a round-robin one. One of the arbiters implementing each policy provides the `ResourceController` interface (annotated as with a '+'")and the other does not.

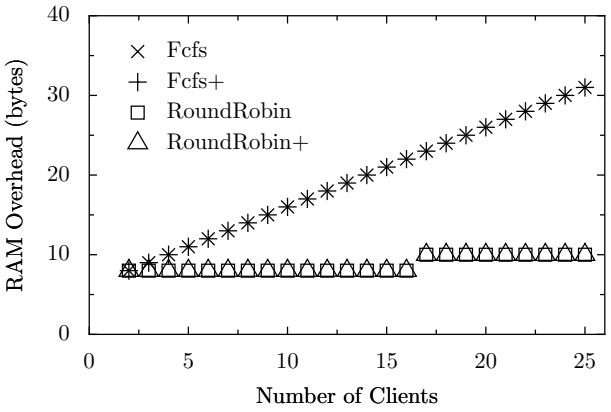

Figure 12: Data memory overhead using an arbiter.

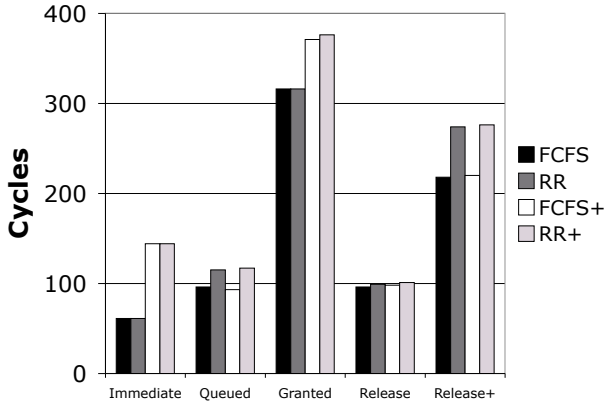

Figure 13: Cycle count overhead of Resource commands using various Arbiters

Figures 11 and 12 show the memory overhead associated with the inclusion of each of these arbiters, and figures 13 and 14 shows the cycle overhead associated with their use.

With only two clients connected, Fcfs arbiters consume approximately 18% fewer bytes of ROM (code memory) than round-robin arbiters, and Simple arbiters consume 14% fewer bytes than normal ones. Approximately 18 additional bytes are consumed per client for a single set of calls to the `request()` and `release()` commands.

All arbiters have the same intrinsic RAM (data memory) overhead of 14 bytes for two clients. Fcfs arbiters require one additional byte for each connected client, while round-robin arbiters only require 2 additional bytes every 16 clients.

Just as one would expect, using the `immediateRequest()` command takes the smallest number of cycles when trying to gain access to a resource. While this operation is very straightforward (if the resource is free, give it to the requesting client), most

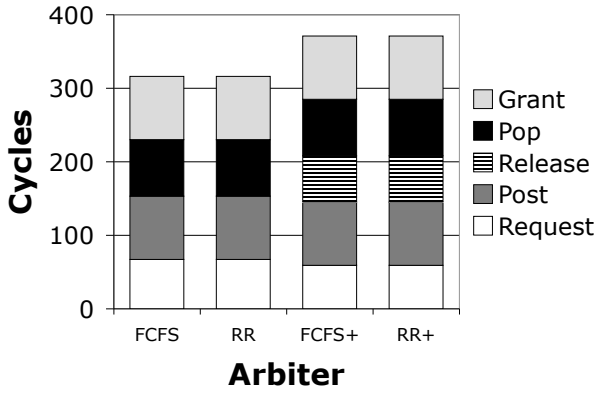

Figure 14: Cycle count overhead when making a non-queing request until a granted event is received

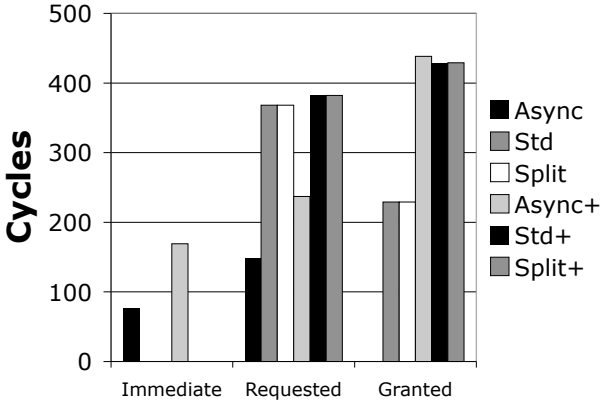

Figure 15: Cycle count overhead of ResourceController implementation by different Power Manager Components

of the other operations are composed of smaller pieces which dominate the time it takes for them to complete. Figure 14 shows an example of the request command broken down into all of its parts. For the "+" arbiters, there is extra overhead associated with the user of the `ResourceContoller` releasing its hold.

In additiion to the arbiters, default power managers are also provided for supporting resources who use three different semantics for starting and stopping them. Ones that have negligible startup time use an "Async" interface, while those with long delays use a "split-phase" one. Figures 15, 16, and 17 summarize the overhead associated with including each of the default implementations provided by T2. The ones annotated with a "+" indicate that they implement the time-delayed feature of powering down the device as outlined in 4.3.1.

7. CONCLUSION

Operating systems are designed to accomplish one

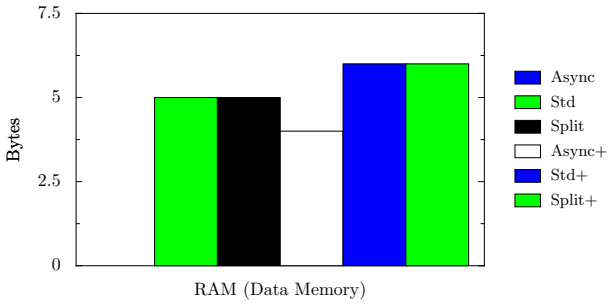

Figure 16: Data memory footprint of each default PowerManager

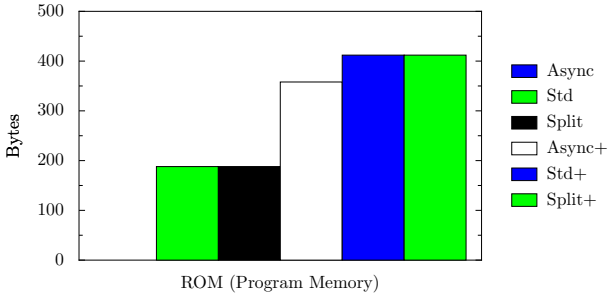

Figure 17: Program memory footprint of each default PowerManager

thing: provide abstractions to underlying hardware resources that make writing programs easier while still maintaining an acceptable level of performance. Determining which set of performance metrics are most important, and building a set of abstractions that can meet the demands imposed by those metrics is a major challenge inherent to operating system design.

We have provided a set of three concrete abstractions for specifically managing memory, energy, and access to peripheral devices. For memory and peripheral devices, a combination of static allocation and compile-time virtualization is used in order to isolate services that share each type of resource from one another. At run time, ownership of these resources is dynamically passed between each service, removing the burden of orchestrating this process from the programmer. By leveraging on the information provided by performing arbitration on a particular peripheral device, the power state it should to be in at any given moment can be determined. Energy management is then accomplished by automatically controlling the power states of both a microcontroller and all of its peripheral devices. Interfaces are provided for performing more fine grained power control or overriding any of the automatic features provided by this abstraction.

Microbenchmark results are presented evaluating each of the different resource management techniques used in an implementation of them for T2. These results are compared against those of several other sensor node operating systems, and the advantages and disadvantages of each in the scope of the specific domain for which they have been designed are discussed.

Acknowledgements

We would like to thank all members of the *TinyOS Core Working Group* for the help they have provided in finalizing the abstractions outlined in this paper. We would especially like to thank those that have co-authored the TinyOS enhancement proposals which document how these abstractions have been implemented in T2.

8. REFERENCES

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

[2] ACPI - Advanced configuration and power interface. `http://www.acpi.info`.

[3] ARM Cortex - M3 Processor. `http://www.arm.com`, Oct. 2004.

[4] V. A. Dinakar Dhurjati, Sumant Kowshik and C. Lattner. Memory Safety Without Runtime Checks or Garbage Collection. In *Proc. Languages Compilers and Tools for Embedded Systems 2003*, San Diego, CA, June 2003.

[5] A. Dunkels, B. Grnvall, , and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors 2004 (IEEE EmNetS-I)*, Nov. 2004.

[6] A. Dunkels, O. Schmidt, T. Voigt, , and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Nov. 2006.

[7] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A dynamic operating system for sensor nodes. In *MobiSYS '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, 2005.

[8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000. TinyOS is available at http://webs.cs.berkeley.edu.

[9] P. Levis, D. Gay, V. Handziski, J.-H.Hauer, B.Greenstein, M.Turon, J.Hui, K.Klues, C.Sharp, R.Szewczyk, J.Polastre, P.Buonadonna, L.Nachman, G.Tolle, D.Culler, and A.Wolisz. T2: A second generation os for embedded sensor networks. Technical Report TKN-05-007, Telecommunication Networks Group, Technische Universität Berlin, Nov. 2005.

[10] MSP430x15x, MSP430x16x, MSP430x161x mixed signal microcontroller. `http://www.ti.com`, Mar. 2005.

[11] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *Proceedings of the Second ACM Conferences on Embedded Networked Sensor Systems (SenSys)*, 2004.

[12] J. Polastre, R. Szewczyk, C. Sharp, and D. Culler. The mote revolution: Low power wireless sensor network devices. In *Hot Chips 16 - A Symposium on High Performance Chips*, Stanford, CA, Aug. 2004.

[13] Intel PXA27x processor family - design guide. Technical Report 280001-002, Intel, May 2005.

[14] J. Regehr, N. Cooprider, W. Archer, and E. Eide. Memory safety and untrusted extensions for TinyOS. Technical Report UUCS-06-007, School of Computing, University of Utah, June 2006.

[15] R. K. Rengaswamy, E. Kohler, and M. Srivastava. Software-based memory protection in sensor nodes. In *Proceedings of the Third Workshop on Embedded Networked Sensors (EmNets 2006)*, Cambridge, MA, May 2006.

[16] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong. A macroscope in the redwoods. In *Proceedings of the Third ACM Conference on Embedded Networked Sensor Systems (SenSys 2005)*, Nov. 2005.