Abstractions for Safe Concurrent Programming in Networked Embedded Systems

William P. McCartney
Electrical and Computer Engineering
Cleveland State University
2121 Euclid Ave, Cleveland OH 44115 USA
w.p.mccartney@csuohio.edu

Nigamanth Sridhar
Electrical and Computer Engineering
Cleveland State University
2121 Euclid Ave, Cleveland OH 44115 USA
n.sridhar1@csuohio.edu

Abstract

Over the last several years, large-scale wireless mote networks have made possible the exploration of a new class of highly-concurrent and highly-distributed applications. As the horizon of what kinds of applications can be built on these networked embedded systems keeps expanding, there is a need to keep the *activity* of programming such systems easy, efficient, and scalable. We make three major contributions in this paper. First, we present a library for TinyOS and nesC that enables true multi-threading on a mote. This library includes support for all mote platforms in use currently (AVR, MSP). Second, we present a tool that can effectively and accurately compute stack requirements for multi-threaded programs. Such analysis ensures that the stacks allocated to individual threads are correctly sized. Finally, we present a collection of programming abstractions that simplifies the construction of concurrent systems for the mote platform. We also present experimental results obtained from several example systems built using our concurrent programming abstractions and the underlying thread library.

Categories and Subject Descriptors

C.3 [**Special purpose and Application-based systems**]: Real-time and embedded systems; D.1.3 [**Concurrent Programming**]: Parallel programming; D.3.3 [**Language Constructs and Features**]: Concurrent programming structures

General Terms

Design, Experimentation, Languages, Measurement, Performance

Keywords

Wireless sensor networks, multi-threading, programming methodology, static analysis

SenSys'06, November 1–3, 2006, Boulder, Colorado, USA.

1 Introduction

In the early years of wireless sensor network (sensornets) design, attention was almost exclusively focussed on programming techniques and methodology that would result in the most efficient software with best performance. Developers and researchers considered software engineering concepts such as ease of programming to be of less importance than performance, small code size, energy efficiency, etc.

In the recent past, however, with sensornet programming architectures maturing well, the focus is shifting to attempts at bringing some of those niceties —programming abstractions, modularity, readability, etc.— into the mainstream of research [11,22,29,30]. The main contributions of this paper are in this domain; we present a set of programming abstractions that enables developers to write concurrent (procedural) programs at the node level in sensornets.

Sensornet programming is typically done in an *event-based* style. Programs are collections of *event handlers* that react to events that the environment presents to a sensor node. The act of *programming* is to then write (short) routines that respond to each of these events. Programs written for *TinyOS* [12], for example, consist of `event`s and `task`s. When there is no event to handle, and there are no tasks to be executed, the mote can enter a low-power mode, thereby conserving scarce energy resources.

This style of programming is great when the program is largely *"modeless"*: there is no state shared among the different tasks and events. However, when there are state dependencies between tasks and events, things get difficult. For example, suppose that node has to send out a set of messages, one to each of its neighbors. The way to do this in TinyOS is to *rip* the operation into two pieces. The `SendMsg.send()` command is a *split-phase* operation; calling it has the effect of starting the send operation, which is completed when the `SendMsg.sendDone()` event is signaled. The program has to explicitly keep track of which messages have been sent out, and which ones have not. Contrast this with the procedural code for doing the same operation: there is a single loop where one message is sent out in each iteration.

In this paper, we present TinyThread, a library that enables multi-threaded programming for TinyOS. TinyThread enables procedural programming on sensor nodes. The library includes a suite of interfaces that provide several blocking I/O operations and synchronization primitives that make concurrent programming both safe and easy. TinyThread

works on all mote platforms that are currently in use (rene, mica family, telos family), and supports TinyOS 1.x as well as the TinyOS 2.0 beta.

In order for multiple threads to run on a mote, it is necessary for each thread to have its own stack in which to store its local state. The common approach to allocating stacks for threads in conventional operating systems is to set aside blocks of memory for each thread. However, given that memory on a mote is a scarce resource, it is important to be precise when deciding on the sizes of individual thread stacks. To solve this problem, we present stack-estimator, a tool that provides tight, yet correct, estimates sizes of thread stacks.

The main contributions of this paper are:

1. TinyThread: a multi-threading library for TinyOS.
2. stack-estimator: a cross-platform stack analysis tool that helps developers to estimate stack depth.
3. A suite of interfaces for blocking I/O operations and synchronization primitives.

The rest of this paper is organized as follows. We compare our work with other similar research efforts in Section 2. Section 3 shows the architecture of our implementation. Section 4 presents the programming abstractions that are included with TinyThread, along with example programs to illustrate the use of these abstractions. In Section 5, we present results from our experiments that measure the impact of using TinyThread in TinyOS programming. In Section 6, we lay out some points about our observations worth noting. After presenting some pointers to future research in this area in Section 7, we conclude with a summary of our contributions in Section 8.

2 Related Work

2.1 Multi-threading and Procedural Code

Y-Threads [23] is a light-weight threading model which separates the thread execution into two stacks. Each Y-Thread has its own stack upon which blocking calls occur, but all Y-Threads share a single stack where non-blocking calls can occur. This can reduce the stack requirements for all of the individual Y-Threads, if all threads only block at the highest levels. Y-Threads may provide an interesting threading model for sensornets as it develops.

Contiki [8] offers a limited form of multi-threading using protothreads. **Protothreads** [7] have the lowest memory requirements of any threading model discussed in this context that can support blocking I/O. While several protothreads can run concurrently, they have a distinct disadvantage compared to traditional threads; protothreads do not save the context of execution. As a result of this, protothreads not only need to use global variables, the global variables must be `volatile`. This can cause some rather poor performance, since the protothread must actually fetch the variable from memory every time it is accessed. For instance, while iterating through a loop, the iteration counter must be stored and retrieved from memory during every iteration. These memory accesses can degrade performance, especially on the *load/store* architectures found on most microcontrollers.

Fibers for TinyOS [29] technically do not require allocating a second stack; there can only be one fiber in the program, and it simply grows the system stack. This threading model cleanly allows users to use blocking I/O calls without the need for a second stack; the limitation is that there can only one *user* fiber. Similar to protothreads, fibers use `setjmp`/`longjmp` for their implementation, but instead of jumping back to the main loop, a blocking fiber call actually executes the scheduler (in some limited form) at the point of execution. This allows users to use local variables and block inside of functions. Many of the blocking I/O routines described later in this paper (Section 4.1) could be ported to support a fiber instead of a thread.

The **MANTIS** [3] OS is a fully multi-threaded operating system for embedded systems. In the MANTIS model, *everything* is a thread. The model of system design and programming is very close to that in conventional operating systems. The fact that everything is a thread, and the fact that any thread can be preempted by another, places some limitations on resource usage. The stack overhead for each thread is considerable. Moreover, the system cannot offer any support to the developer in detecting problems with concurrency of the nature that TinyOS/nesC can. By contrast, TinyThread provides the flexibility of multiple threads, while at the same time, by operating within the confines of the concurrency rules in TinyOS, preserves the concurrency model of TinyOS. Moreover, TinyThread does not prevent the developer from using event handlers in addition to the procedural multi-threaded; this is not feasible in MANTIS.

Maté [17] is a virtual machine environment for TinyOS. Although in general, the design goals of Maté are quite different from those of TinyThread, there are some similarities worth noting. Like TinyThread, Maté also provides a way for treating split-phase operations as though they were straight-line pseudo-blocking operations. However, since Maté is a virtual machine, these operations are implemented through byte-code operations, and hence incur the cost of byte-code interpretation.

2.2 Stack Analysis

Basic stack analysis in embedded systems has been done by simply measuring constant addition to the stack register [4]. This approach tends to overestimate, unless several constraints are put on the developer. On some platforms, this constant-only approach simply is not possible. Stack analyzers try to determine the worst-case stack usage to avoid overrunning the allocated space.

Stacktool [25] provides a method of analysis by actually monitoring the values passed explicitly to the general purpose registers and then out to special purpose registers, such as the stack pointer. Stacktool performs this analysis on a compiled binary file for the ATmel AVR platforms. It internally disassembles the machine code before it processes it. Stack analyzers are traditionally tied to one platform or another, although the techniques are usually more general. We compare our stack-estimator with stacktool in Section 3.2.
HOIST [24] builds a majority of platform-specific points required for stack analysis. Hoist accomplishes this by using a processor (or simulator) as a black box. It is currently limited to 8-bit processors.

2.3 Programming Abstractions

In the recent past, there has been a considerable amount of work on developing usable programming abstractions for sensornet development. **Hood** [30] is an abstraction that allows a node in a sensornet to easily access and interact with other nodes in its neighborhood. The Hood abstraction allows a node to easily share its local state with neighboring nodes. What this means under the covers is that there are several concurrent activities being performed: each node keeps track of its state, sends out data periodically, and receives data from other nodes and filters them for use. An implementation of Hood using the TinyThread abstractions could simplify these activities and the concurrency introduced. Similar to Hood, **abstract regions** [29] provides access to high-level abstractions such as *N-radio hop*, *k-best neighbor*, etc.

The programming API that we present as part of TinyThread is similar in spirit. The goal is reduce the number of low-level operations in a sensornet program. Our abstractions have to do with synchronization among concurrent threads in a program.

TinyRPC [20] is a remote procedure call interface for TinyOS. Using TinyRPC, a component can bind to some interface that is actually implemented in a remote node in its neighborhood. TinyRPC supports both named bindings and discovered bindings between nodes. Once the binding has been set up, it is easy for nodes to communicate without having to think about messages, node ids, etc. TinyThread provides an exciting new use for TinyRPC. Since nodes can issue remote calls, that means that two threads running on two separate nodes can now use our barrier synchronization abstract to communicate and synchronize directly.

3 TinyThread Architecture

In this section, we present the internals of TinyThread. TinyThread is written in *nesC* for *TinyOS*. TinyThread runs on all AVR (mica, mica2, micaz) and MSP430 (telos, tmote) platforms, with preliminary support for ARM/PXA (imote) platforms. TinyThread is implemented as a library on top of an event driven OS (in this case TinyOS), allowing the intermixing of event-driven programming and threaded programming. Layered on top of TinyThread is a set of blocking I/O routines and synchronization primitives.

3.1 Types of threading

The term *thread* refers to a context of execution in a program. In a programming system that supports *multi-threading*, several threads can be programmed to run concurrently. On a uni-processor computer, however, only one thread can actually execute at a given time. The remaining threads are said to be *ready*. On modern operating systems, each thread utilizes a *stack* to store the current context of the processor while it is *ready*. The next time a *ready* thread is allowed to execute, the context is taken from the stack, and the thread proceeds from where it left off. The decision of which thread is currently *running* is commonly made by a *thread scheduler*, which allocates *time slices* to each thread.

A *task* in TinyOS is sometimes considered to be an extremely lightweight thread. This can be misleading; a task has no resident context of execution. When a task terminates, its context of execution is lost. TinyOS tasks always execute to completion. They can only be preempted by interrupts, called *async events*, and by callbacks, known as *signals*, which are explicitly called by said task. This simple system of tasks running to completion and a system of callbacks is central to *event-based programming*. Tasks provide no way for one task to preempt another.

3.1.1 Stack-less Threading

It is possible to implement extremely light-weight thread-like structures without the use of a thread stack [7]. This stack-less operation is implemented using the functions `setjmp` and `longjmp`. These routines are the base for *user-based threads*, but as Engelschall [9] points out, this only solves the easy half of the thread problem. The harder half of the problem lies in storing the processor state. In an embedded system without an MMU, this amounts to storing the registers and swapping to another stack.

If the stack of the thread is not stored, there are several constraints faced by the developer. First, any local variables used in a thread can be corrupted or overwritten during a *blocking* call. An easy solution to this problem is to simply use global or `static` variables. However, never using local variables is a bad idea with respect to encapsulation and modularity, and makes programs prone to errors. Such a practice also has further complications: the code may execute correctly in a task, but fail in a stack-less thread.

The second major limitation is that a stack-less thread cannot block inside of a routine; in order to block, `longjmp` must be called inside of the top level of the thread. This removes much of the ability of developers to layer abstractions on top of these primitives. However, the programmatic difficulties added by utilizing these stack-less threads may be offset by the ability to use blocking I/O. This may be an acceptable tradeoff for certain applications, but it competes against writing modular, maintainable applications.

3.1.2 Threads in TinyThread

Threads in TinyThread follow the traditional definition; each thread has its own stack, thereby enabling it to store its entire execution context when *ready*, and then to reload the context when re-enabled. This is the definition of thread we will use in the rest of this paper.

There are two main forms of multithreading: *cooperative* and *preemptive*. Cooperative multi-threading is the simpler of the two. Threads still operate with their own context, but the only way for a thread to stop executing is by explicitly yielding its execution via a call to `yield()` or implicitly via a blocking I/O routine. This essentially means that cooperative threads can run with implicit locking [7]. Protothreads and TinyOS fibers are also examples of cooperative multi-threading. Some cooperative multi-threading implementations only allow yielding to a specified (say, system) thread, while others allow developers to simply yield to a scheduler. In an embedded system, cooperative threads run with implicit locking, which means that threads are not preempted by other threads. Instead, a thread implicitly controls when it can be swapped out. This implicit locking is a double-edged sword. It allows developers a clear view of exactly what variables need to be protected, but on the other hand, it forces developers to be aware of the time any computation might take, to avoid blocking the processor for too long.

Execution Model	Blocking I/O	Implicit Locking	Local Variables	Multiple Threads
TinyOS Tasks		✓	✓	
ProtoThreads	✓	✓		✓
TinyOS Fibers	✓	✓	✓	
Cooperative Multithreading	✓	✓	✓	✓
Preemptive Multithreading	✓		✓	✓

Figure 1. An overview of capabilities of various kinds of concurrency models

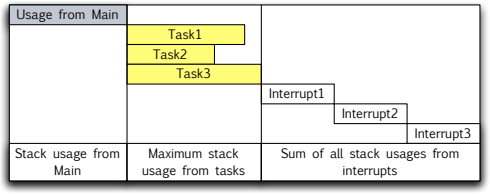


Figure 2. Naïve calculation of stack usage: sum of the interrupts' stacks and the maximum task stack usage

Forcing developers to be fully aware of the length of time a set of library routines takes is rather cumbersome. Take, for instance, calculating triple DES on a block of data. It is near impossible to guess at how long the calculation may take without some previous benchmarks. Preemptive multi-threading can solve this problem by forcibly preempting a running thread after some finite period of time. However, such arbitrary preemption can create concurrency problems, since a thread may be preempted in the middle of a critical section. Preemptive multi-threading can also overcome deadlock. For instance, if a task or cooperative thread goes into an endless loop, the system is deadlocked. Preemptive threads do not deadlock a system when they themselves go into an endless loop. A breakdown of the different features supported in these threading models is shown in Figure 1.

3.2 Stack Size

Given that each thread has to keep a separate stack to store its context, how big should each stack be? In conventional operating systems, this is not a problem: each thread is given a stack of some standard (usually large) size. Such generous stack depth allocation does not make sense in the mote context, given the severe constraints on available resources.

Currently, TinyOS has no built-in tools to compute stack size; there is no need, since developers never need to allocate the stack size manually. In other systems such as MANTIS [3], thread stack size is allocated by using an "educated guess" on the part of the developer. Such guessing is unsafe, since a wrong estimate of stack size could lead to problems — a guess that is too low could lead the program to crash, and a guess that is too high wastes memory resources (which are scarce to begin with). There are tools which can calculate the stack requirement for some microcontrollers, but they require special knowledge of the operating system if any threading model is used.

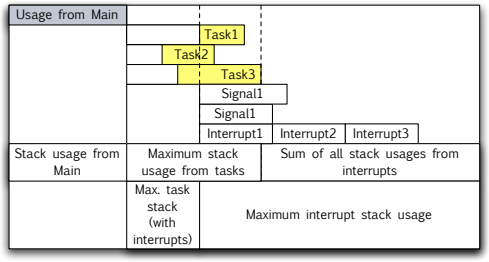


Figure 3. Context sensitive stack usage calculations: adding all the interrupts' stack usages to each other only if and/or where interrupts are enabled

The TinyThread tool suite includes a tool (called stack-estimator) that analyzes stack usage of TinyOS applications for all platforms that TinyThread supports. In order to analyze stack requirements, it is the worst-case scenario stack that must be allocated. We use an approach similar to that described by [27] in a different type of call graph. Static stack analysis has been explained, and even demonstrated for TinyOS by Regehr *et al.* [25].

Static stack analysis of TinyOS applications can be done in a very simple way. The total stack depth required is simply the sum of (i) the stack usage of the `main()` function, (ii) the maximum stack usage by any task, and (iii) the sum of stack usages for each interrupt handler:

$$sd_{total} = sd(main) + Max_{i=1}^{n} sd(task_i) + \sum_{j=1}^{n} sd(int_j)$$

This simple, naïve approach to stack estimation, shown in Figure 2, although safe, may allocate more stack space than can possibly be required, resulting in wastage of memory.

A better way to calculate stack depth is by using a context-sensitive static analysis [25]. The essence of this strategy is to exploit the fact that interrupts are *disabled* in different parts of the application. In TinyOS, this extends far beyond simply inside of events, but continues through the interrupt handlers themselves. GCC supports two different kinds of interrupt handlers: *interrupts* are interrupt handlers that have interrupts enabled, and *signals* are interrupt handlers with interrupts disabled. The latter provides an extreme amount of stack savings, since hardware *signals* cannot preempt other hardware *signals* or *interrupts*. For reference, in TinyOS 1.1.x, in MSP430-based platforms, all interrupt handlers are *signals*, giving excellent stack savings. An overview of the calculations made can be found in Figure 3, and can be summarized as follows:

$$sd_{total} = sd(main) + Max_{i=1}^{n} sd(task_i[int\_enabled]) + Max(sd(interrupt_{wc}), Max_{k=1}^{n} sd(signal_k))$$

In the equation above, $task_i[int\_enabled]$ refers to the portion of $task_i$ in which interrupts may occur (they are not disabled). Rather than just taking the sum of stack depths of all the interrupts, we now take the worst-case sum of stack usage for all interrupts, which is a lower number than the plain sum, since inside of some interrupt handler(s), other interrupts may be disabled.

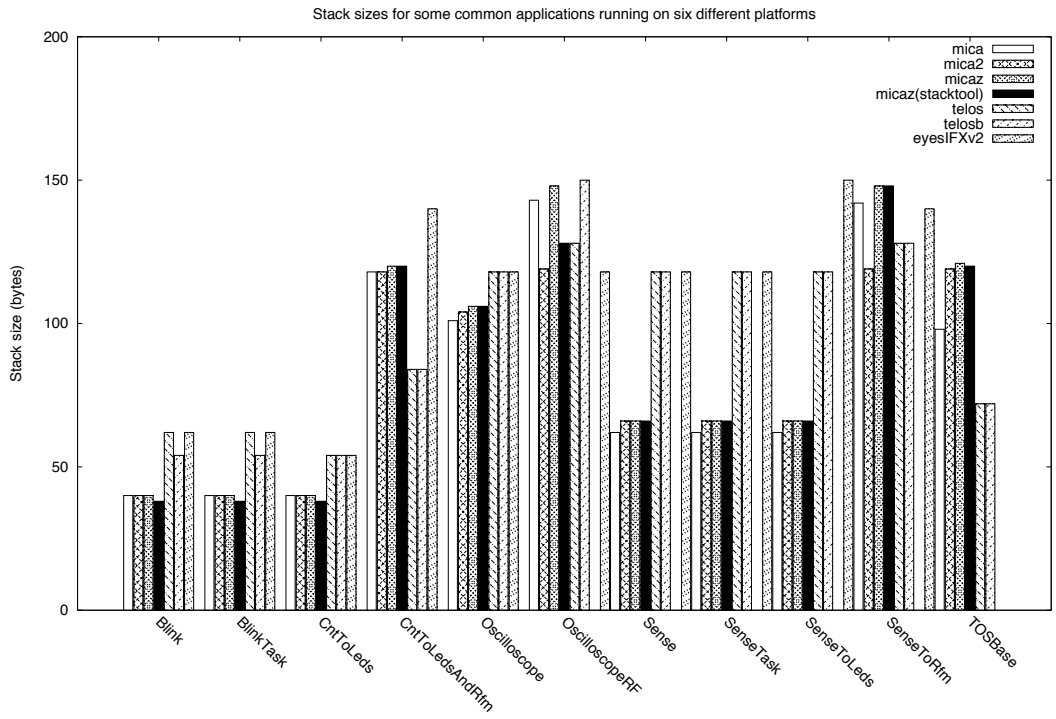


Figure 4. Stack size estimates for a few applications from the `tinyos/apps` directory in the TinyOS distribution. The plot shows the result of our stack analysis tool (stack-estimator) run for the mica, mica2, micaz, telos, telosb, and the eyesIFX2 platforms. It also shows the result of stacktool [25] run on the micaz platform.

The stack analysis routine provided TinyThread supports several platforms, and can estimate the maximum possible stack size required for any TinyOS application. Figure 4 shows the stack requirements for many of the applications in the TinyOS `apps` directory. The chart also shows the results of using stacktool [25] on micaz. TinyThread follows the approach laid out by stacktool, by simulating read/writes to registers. Instead of disassembling the instructions internally, stack-estimator uses *objdump* to do the disassembly. As shown in the figure, the two tools are nearly identical.

This stack analysis program generates a header (stacksize.h) which contains a set of `#define`s which can be used to allocate the stacks for each particular thread. This provides a statically allocated stack for the thread, without wasting any extra resources.

Finding the necessary stack size for a given thread in a program is a two step process. To start with, the stacks can be declared to be of arbitrary length. Our stack analysis tool operates on the binary image of the program. Once the program has been compiled for a particular platform, we can run stack-estimator to compute stack usage for the application. Figure 5 shows the stacksize.h file generated for the Gossip application (Listing 7). These constants can now be used in the program to declare the stacks of the maximum length that the threads would actually need.

```
#define  threadInitIdle_STACKSIZE  88
#define  __ctors_end-0x3a_STACKSIZE  56
#define  SendAckTask_STACKSIZE  76
#define  PacketRcvd_STACKSIZE  70
#define  threadActive_STACKSIZE  88
#define  thread_wrapper_STACKSIZE  58
#define  startSend_STACKSIZE  72
#define  signalRXFIFO_STACKSIZE  84
#define  signalTXFIFO_STACKSIZE  76
#define  thread_task_STACKSIZE  66
#define  threadNInitIdle_STACKSIZE  88
#define  taskInitDone_STACKSIZE  56
#define  send_task_STACKSIZE  72
```

Figure 5. An excerpt of the stacksize.h file generated by stack-estimator for Gossip. The file contains named constants for the stack usage of every part of the program.

3.3 TinyThread Scheduler

The core of TinyThread is a very simple FIFO thread scheduler. The scheduler itself runs inside of a regular TinyOS task. The scheduler maintains a list of threads in the program. Each thread can be in one of four states: *ready*, *sleeping*, *blocked*, and *locked*. These states and the transitions allowed among them are shown in Figure 6. However, for the purpose of scheduling, we can group the latter three states all into an *inactive* state. When the scheduler task is posted, and is ready to execute, the thread scheduler walks down its list of threads until it finds a thread in the *ready* state. When a *ready* thread T_k is found, the TinyThread scheduler swaps the TinyOS system stack with the T_k's stack. This way, the context of wherever T_k "left off" is regained.

The thread T_k now runs (a) until completion, or (b) until it executes some blocking call. In either case, the thread stops executing and is no longer in the *ready* state. The thread

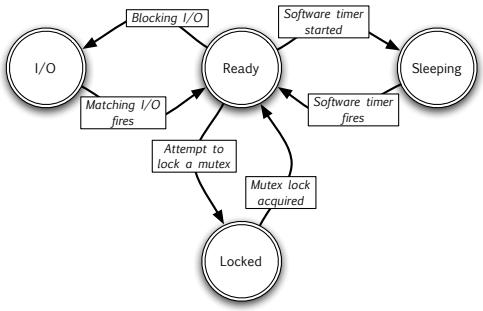


Figure 6. Thread states and transitions among them

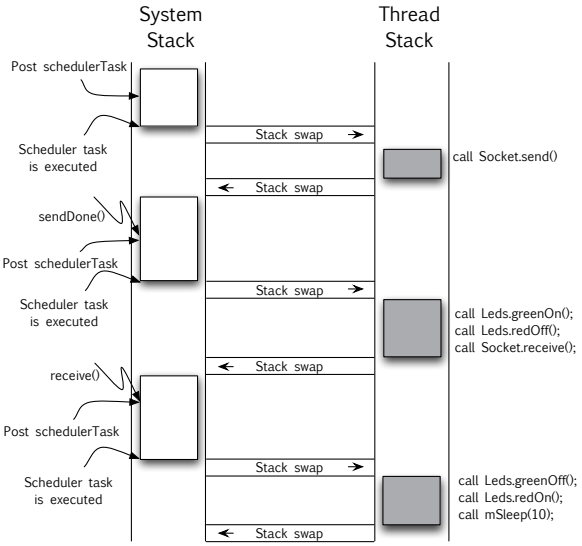


Figure 7. Illustration of an application running in the TinyThread scheduler. The white boxes show normal TinyOS operation. When the thread scheduler is executed, the first thing that happens is that the system stack is swapped out and the thread stack takes over. The gray boxes show code executing inside of a thread.

scheduler now swaps the stacks back to restore the TinyOS system stack. Before it yields, however, the thread scheduler needs to figure out if it must post itself again or not. This is determined by whether there are any more *ready* threads. If the scheduler finds some *ready* thread, it re-posts itself. If there is no thread in the *ready* state, *the scheduler does not re-post itself*. If this were not the case, and the scheduler always re-posted itself, that would be a waste of processor resources, and consequently, a waste of power.

Every time the scheduler task is executed, exactly one thread in the thread pool runs. So if there were *n ready* threads in the thread pool, the thread scheduler must be allowed to execute (by the TinyOS task scheduler) *n* separate times. Although TinyThread does support preemption of threads by other threads, preemption is turned off by default.

Figure 7 shows the execution of an application with one thread. Initially, the thread scheduler task is posted, and will eventually get a chance to execute, controlled by the TinyOS scheduler. At this point, the TinyThread scheduler sees that there is one *ready* thread, and hence swaps stacks, and begins to execute the thread. The first statement in the thread code is a call to `Socket.send()`, which turns out to be a blocking call (Section 4.1). The TinyThread scheduler now puts the thread in the *inactive* (*blocked*) state, and sees if there are any other *ready* threads in the thread pool. Since there is only one thread in the pool, and that thread is not *active*, the thread scheduler does not re-post itself. It simply yields to the TinyOS task scheduler.

Eventually, the `sendDone()` event corresponding to the `send()` command is signaled. At this point, the event handler for this event posts the thread scheduler task. The process described above repeats for the program's lifetime.

The key thing to note here is that the TinyThread scheduler is extremely sensitive to power utilization. The scheduler task runs *only when there is at least one* ready *thread*. The rest of the time, the thread scheduler is sleeping, allowing the mote to sleep and conserve power as well.

3.4 Context Switching

A context switch stores all of the working registers (the state of the processor) onto the currently executing stack, switches the stack pointer to the new stack, and restores the working registers from the new stack (shown in Listing 1). Specifically, first the general purpose registers, and the status registers are pushed onto the current stack. The stack pointer is then swapped to a different stack pointer. The status register is popped from the stack followed by the general purpose registers in reverse order. These actions occur inside of a function which stores the return address when invoked. So the final step, the function's return instruction, pops the return address off of the new stack, allowing for seamless resuming of the previous context. The context switching routines were written for gcc using inline assembly.

3.5 Implementation Platform Support

At the time of writing this paper, there are two versions of TinyThread: one for the current TinyOS 1.x distribution (1.1.15), and another for the current TinyOS 2.0 beta. The nice thing is that the two versions provide (almost) the same API to developers[1]. Since the API is the same, applications using TinyThread extensively are easily (almost trivially) ported between TinyOS 1.x and TinyOS 2.0. The wiring files for the two versions differ radically, as each version is consistent with the style of the respective TinyOS version (*e.g.,* using generic objects over unique statements). Both of these versions include support for all MSP430- and AVR-based platforms.

3.6 Programming using TinyThread

Threads in TinyThread are created using the `create_thread()` command. This command takes as parameters a pointer to the function that will run in the thread, and a portion of memory to be used as the stack for the thread. Listing 2 shows an example of one of the threads in the Gossip application (from Section 4.2.2) being created.

[1]Some calls in the API are different, in order to conform with the TinyOS 2.x conventions. For example, the order of parameters to `SendMsg.send()` is different than in TinyOS 1.x.

Listing 1. Actions performed when switching stacks.

```
1 // subroutine call pushes the PC onto stack
2 yield() {
3   PUSH_GPR();
4   PUSH_STATUS();
5   SWAP_STACK_PTR();
6   POP_STATUS();
7   POP_GPR();
8 }
9 //return, which pops the PC off the stack
```

Listing 2. Declaring and creating a thread in TinyThread

```
1 // Declare stack for thread
2 uint16_t activeStack[threadActive_STACKSIZE];
3 ...
4 // Define the function that will run in the thread
5  void threadActive() { ... }
6 ...
7 // Create the thread (upon initialization)
8 call create_thread(threadActive, stack_top(activeStack,
9                     sizeof(activeStack)));
```

The stack passed to `create_thread()` is used by the thread to store its execution state when it is not running. The actual implementation of the function that will run in the thread is written in procedural style. Examples are presented in the next section. The size of the stack (line 2) comes from the constants file (stacksize.h) that stack-estimator generates.

4 Programming Abstractions

In order to make the TinyThread library useful to developers immediately, we have implemented a few programming abstractions that can greatly help in writing safe concurrent programs. We present some of these abstractions here. We have implemented several applications (about 20) using TinyThread and the accompanying API. Some of these applications are presented as examples throughout this section and the rest of the paper.

4.1 Blocking I/O

One of the primary causes for stack ripping in purely event-based programming is the inability to perform blocking I/O operations. When programming sequential enterprise systems, it is natural to use a `read x;` statement, and simply block execution until `x` becomes available. However, in a sensornets context, such blocking is dangerous; the program may miss important events while the processor is blocked.

TinyThread provides blocking versions of many commonly-used I/O operations to access the sensors and the radio on the mote. Since these operations block, they can only be used from inside of a thread; never directly in a task or event handler. The `BlockingADC` interface provides the `readADC()` function (Lines 1–3 in Listing 3). When called, this function simply blocks the thread until the data from the ADC becomes available. In the very next line of code, the program can actually use the data collected from the sensor.

Example 1: Oscilloscope

The Oscilloscope application that is distributed as one of the samples with TinyOS is a simple application that samples sensors on a mote, and sends this sampled data via the UART to a PC. The PC can then visually render the data that

Listing 3. Blocking I/O using TinyThread**.**

```
1 interface BlockingADC {
2   command uint16_t readADC();
3 }
4
5 interface Socket {
6   command result_t send(uint16_t address, uint8_t length,
7                         TOS_MsgPtr msg);
8   command result_t receive(TOS_MsgPtr m);
9 }
```

Listing 4. Implementation of the `Oscilloscope` application using threads. The `Socket Send()` routine is wired to a fully buffered fifo sending queue.

```
1  void oscope_thread() {
2    struct OscopeMsg *pack;
3    uint16_t reading;
4    uint8_t i;
5
6    while (TRUE) {
7      for(i=0;i<BUFFER_SIZE;i++) {
8        //Read sample
9        reading = call readADC();
10       if (reading > 0x0300) call Leds.redOn();
11       else call Leds.redOff();
12       pack = (struct OscopeMsg *)msg.data;
13       pack->data[i] = reading;
14       call mSleep(125);
15     }
16     pack->channel = 1;
17     pack->sourceMoteID = TOS_LOCAL_ADDRESS;
18     r = call Socket.send(TOS_UART_ADDR,
19                          sizeof(struct OscopeMsg), &msg);
20     call Leds.yellowToggle();
21   }
22 }
```

the mote senses. This program responds to three events: (*i*) a Timer that fires every 125 ms, (*ii*) a notification from the ADC interface that data is ready, and (*iii*) a notification from the SendMsg interface that the message has been queued for transmission. Every time the timer fires, the `ADC.getData()` command is called to read a sample from the sensor. Then when the data becomes available, a task is posted that actually sends the message over UART. The result is that the functional code in the program is split over three separate functions, making it hard to read.

Contrast this with our implementation written using TinyThread's blocking I/O operations, shown in Listing 4. The logic of the program is now much easier to see, and there is no need for manually ripping the function into pieces.

The `Socket` interface (Lines 5–9 in Listing 3) provides blocking operations for sending and receiving messages. The `send()` operation in `Socket` blocks until the message has actually been sent (until the `sendDone()` event is raised). This means that if a node needs to send a series of messages, they can actually be sent out in a loop rather than ripped apart in several functions. The `receive()` operation simply blocks until there is a message waiting to be processed.

Example 2: Bounce

This is a very simple application intended to illustrate the use of the blocking receive operation (Listing 5). The appli-

Listing 5. Implementation of the `Bounce` application using threads. This example illustrates the blocking receive functionality in `Socket`.

```
1  void threadBounce() {
2    while (TRUE) {
3      call mSleep(100);
4      call Socket.send(!TOS_LOCAL_ADDRESS, 2, &mymsg);
5      call Leds.greenOn();
6      call Leds.redOff();
7      call Socket.receive(&mymsg); // Block until message
8      call Leds.redOn();
9      call Leds.greenOff();
10   }
11 }
```

Listing 6. Synchronization using TinyThread.

```
1  interface Mutex {
2    command void lock(mutex * m);
3    command void unlock(mutex * m);
4  }
5
6  interface Barrier {
7    command void block();
8    command void unBlock();
9    command void checkIn();
10 }
```

Listing 7. Implementation of a Gossip diffusing computation through a network. The two threads in the program synchronize using a pair of barriers.

```
1  void threadIdle() {
2    int i;
3    GossipMsg *rMessage;
4    call Socket.receive(&msg); // Wait for the first message
5    rMessage = (GossipMsg *) msg.data;
6    parent = rMessage->source;
7    state = ACTIVE;
8    atomic call Leds.redOn();
9    call ActiveBarrier.unblock();
10   for (i=0; i<n_nbrs; i++)
11     if (neighbors[i] != parent)
12           sendGossipMessage(neighbors[i]);
13   call CompleteBarrier.block();
14   state = COMPLETE;
15   atomic { call Leds.redOff(); call Leds.greenOn(); }
16   sendGossipMessage(parent);
17 }
18
19 void threadActive() {
20   int i;
21   call ActiveBarrier.block();
22   for (i=0; i<n_nbrs; i++) call Socket.receive(&mymsg);
23   call CompleteBarrier.unblock();
24 }
```

cation runs on two motes, and two motes continually bounce a message back and forth. When a mote receives a message, it turns on its red LED, and when it sends a message, it turns on its green LED. The interesting piece in this example is on line 7, where the program simply waits for the next message. The thread blocks until a message actually arrives.

4.2 Synchronization

4.2.1 Mutex

The simplest synchronization primitive that TinyThread provides is a way for threads to acquire mutually exclusive access to some critical section. The `Mutex` interface (Lines 1–4 in Listing 6) allows a thread to `lock()` a mutex and enter its critical section. Once it is done executing its critical section, the thread calls `unlock()` to relinquish critical section access to another thread that wants to use it.

4.2.2 Barrier Synchronization

A *barrier* is a primitive for *rendez vous* synchronization among a set of threads [10]. The `Barrier` interface (Lines 6–10 in Listing 6) in the TinyThread library provides two kinds of barrier synchronization. First, it supports pair-wise synchronization between two threads. The thread that arrives at the barrier first calls the `block()` command. This causes this thread to stay *blocked* until it is woken up by the other thread. When the second thread has also arrived at the barrier, it calls `unBlock()`, causing both threads to be *ready*.

Example 3: Data diffusion in a network

Data diffusion protocols are used frequently in sensornets, in a wide variety of ways [13,14,18]. Consider a simple Gossip data diffusion protocol in a network. Some node initiates the "gossip", which is propagated through the network until it reaches the edge of the network. When a node receives a gossip message for the first time, it marks the neighbor who sent it as its *parent*. When a node does not have any neighbors other than its parent, it sends the message back to its parent. When a node has heard from all of its (non-parent) neighbors, it sends the message back to its own parent. In this manner, the initiating node eventually receives acknowledgment that the entire network has seen the message.

Listing 7 shows the multi-threaded implementation of this protocol. The two threads—`threadIdle` and `threadActive`—are started at the same time. But the second thread blocks until the node enters the `ACTIVE` state (the node has received its first gossip message, and is now active in propagating the message). The former thread, `threadIdle` is also blocked until the first message arrives. Once the first message arrives, and the node transitions into the `ACTIVE` state, `threadIdle` has also arrived at the barrier that `threadActive` is waiting at, and calls `ActiveBarrier.unBlock()` to signal the rendez vous. At this point, both threads are active.

`threadIdle` now sends out the gossip to all neighbors (except the parent). After this, it blocks on another barrier — `CompleteBarrier`, waiting until all neighbors have responded to the gossip message. The `threadActive` thread waits until it has received acks from each one of its neighbors, and when all acks have been received, it unblocks the `CompleteBarrier`, and with it the other thread.

In addition to pair-wise barrier synchronization, the `Barrier` interface also supports synchronization of a number of peer threads. In this case, when each thread arrives at the barrier, it calls the `Barrier.checkIn()` command. When the last thread to arrive at the barrier calls `checkIn()`, all threads that are blocked at the barrier are woken up at the same time. Using this primitive, all the threads may take a step (say, a state reconfiguration) together. Moreover, if these threads were running on different nodes, the barrier can be used for synchronization at the network level.

4.2.3 Rendez Vous-based Message Passing

The TinyThread API can be extended to implement richer synchronization actions. The current implementation of `Socket` is one that is layered on top of `AMStandard`. According to the semantics of `AMStandard`, when a node sends a message using `SendMsg.send()`, the corresponding `SendMsg.sendDone()` event tells the node that the message has been successfully placed in the active message buffer. The node does not know if the message actually arrived at its destination; that is too much information.

However, consider a different implementation of `Socket` that employs a acknowledgment scheme to confirm that a message actually arrived at the destination. In such an implementation, the `send()` operation completes only when the sender has received the ack message from the destination. In this case, the sender thread is blocked until the destination is ready to receive the message.

Note: TinyThread is a *"lightweight"* extension to nesC. The existence of the thread library and the programming primitives presented here do not in any way preclude developers from using regular TinyOS/nesC programming constructs and idioms. For instance, inside of a thread, an event can be signaled without incident. The only exceptions have to do with preemption and are explained in 6.2.

5 Evaluation and Results

Experimental Setup: We ran our experiments in our lab on Tmote Sky and micaZ motes. We also tested TinyThread on mica2 and mica2dot motes, although all the measurements presented here are from Tmote and micaZ motes. Applications that had Java interfaces ran on a Pentium IV (2.8 GHz) PC with 1 GB of RAM. The power measurements were recorded using a Tektronix oscilloscope connected to a mote.

5.1 The Cost of Multi-Threading

What is the true cost of multi-threading? In doing our research on building TinyThread, we developed an analysis model that can be used to measure the true cost of multi-threading for embedded systems. In systems with a simple processor architecture, such as in embedded systems, the main building block for enabling multi-threading is a way of changing the stack used for the thread that is executing.

This stack-swapping operation is also the primary contributor to the cost of the multi-threading implementation. There are two kinds of costs that one has to pay in order to get multi-threading. The first is that multi-threading incurs a degradation in performance. The performance overhead is caused by the extra instructions that have to be executed in order to swap the stacks. This is a finite number that can be statically determined, and depends on the processor.

The second cost is memory — each thread has to be allocated its own stack, which in turn has to be allocated in the RAM. This RAM overhead is finite, and can be statically determined [25]. The RAM overhead itself has two contributing factors. The first is the overhead caused by interrupt handlers, and is dependent on which operating system is being used. The second piece to the RAM overhead is size of the program context, and this is dependent on the processor.

In addition to the overhead of the threads themselves, there may be additional costs, such as overhead introduced by any ancillary routines that the library may provide. This cost entirely depends on what routines are provided. The cost of our own blocking I/O routines is extremely small.

5.2 Context Switching Costs

As shown in Section 3.4, any time a context switch occurs in TinyThread there is a definitive cost which occurs. On the MSP430 processor [26], each stack-swapping operation amounts to 33 instructions (66 instructions including the swap back to the system stack). In terms of actual execution time, this translates to 196 processor execution cycles. On the AVR ATMega processors [2], the overhead is higher: the stack-swapping takes 168 instructions for the swap to the new thread and back (total of 332 cycles of actual execution time). This cost is added to the time it takes for TinyOS to schedule the task in which the TinyThread scheduler runs. In addition to this finite overhead, when the TinyThread scheduler begins to execute, it has to walk through its list of threads to see which of those is *ready*. The context switching costs affect response time the most. Every blocking call will switch contexts into the system thread until the call unblocks.

5.3 Resource Usage

To test the resource usage, four applications were compiled and their resource consumptions analyzed. The three metrics used are stack consumption, RAM usage and ROM (program space) usage. The three most popular platforms that TinyOS supports were used during these experiments.

TinyThread also provides a set of compile time options which allows the user to enable/disable certain features. One option available is the maximum number of threads in the scheduler. Since threads can be created at run-time, a maximum number of threads must be enforced on the scheduler. If no thread count is specified, it is assumed that the maximum is four. Even though most applications will not use four threads, it is better to provide extra resources which developers can compile out later on as required.

Another compile time option is thread-safe I/O. This makes it safe for multiple threads to access the same I/O primitive simultaneously. For many I/O primitives, they are always thread safe, but `Socket.send` for instance is not. The effect of turning this off is not catastrophic, as it is on a socket in other operating systems. It simply causes threads to contend, using extra battery when multiple threads are trying to send messages at the same time. Thread safety is enabled by default, but can be disabled at compile time.

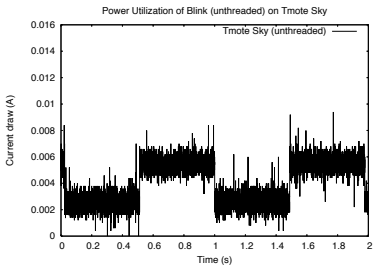
We ran tests on four applications Blink, Bounce, Filter, and SimpleComm. All four of these applications use only one thread. We took extra care to make sure that the actions performed in the threaded version of the application directly matched those in the event-driven version. This is to rule out "savings" in cost caused by programmer intelligence, and only measure the actual technique described.

Table 1 summarizes the resource analysis of the TinyThread version compared to the event-driven version. This table uses only the ROM usage and the total RAM usage. The total RAM usage is derived from the stack usage of the application and RAM usage of the application:

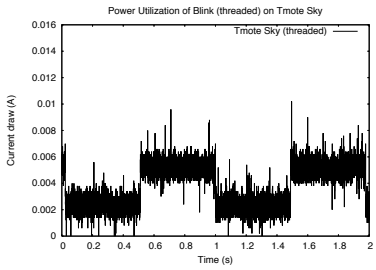
$$TotalRAMUsage = StackUsage + DataSpaceUsage$$

		Threaded Version		Event Driven Version		Difference	
Platform	Application	RAM	ROM	RAM	ROM	RAM	ROM
telosb	Blink	164	3226	94	2610	70(42%)	616(19%)
telosb	Bounce	587	12746	461	11834	126(21%)	912(7%)
telosb	Filter	2741	12850	2507	11840	234(8%)	1010(7%)
telosb	SimpleComm	641	12578	389	11632	252(39%)	946(7%)
mica2	Blink	193	2534	89	1610	104(53%)	924(36%)
mica2	Bounce	830	12402	569	10860	261(31%)	1542(12%)
mica2	Filter	3067	12668	2626	11088	441(14%)	1580(12%)
mica2	SimpleComm	911	12156	511	10656	400(43%)	1500(12%)
micaz	Blink	193	2564	89	1640	104(53%)	924(36%)
micaz	Bounce	834	11756	515	10038	319(38%)	1718(14%)
micaz	Filter	3100	11994	2571	10258	529(17%)	1736(14%)
micaz	SimpleComm	949	11548	454	9844	495(52%)	1704(14%)

Table 1. Overall Resource Consumption: Threaded Versus Event Driven.



(a) Power draw of a Tmote Sky running Blink (unthreaded).



(b) Power draw of a Tmote Sky running Blink (threaded).

Figure 8. Comparison of power utilization of Blink running on a Tmote Sky mote. The figure on the left shows the current draw of the mote running the application without TinyThread, and the plot on the right is with TinyThread.

Data space is generally allocated at the lowest available memory location, while the stack generally grows from the highest memory address down. A stack overflow occurs when the top of data memory is about the lowest used stack position. For TinyThread the individual thread stacks are allocated in the data space already.

5.4 Power Utilization

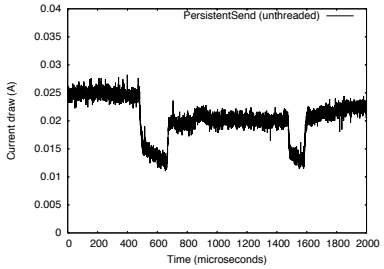
Using TinyThread to develop applications *appears* to fundamentally change the way applications execute on the mote. The presence of blocking operations for using sensors and the radio seems to suggest that there may some inefficiency in the amount of power utilized by the mote. This is not the case. The TinyThread scheduler is extremely sensitive to power utilization, and in almost all cases, the power draw of a program written using TinyThread is exactly the same as the corresponding program written using regular TinyOS tasks and events.

We ran power utilization tests on several applications. The first one is the most simple application of all: Blink. Figure 8 is a comparison of the current draw in a Tmote Sky mote running two versions of Blink. As Figures 8(a) and 8(b) show, there is no change in the amount of current drawn in the mote as a result of using TinyThread.

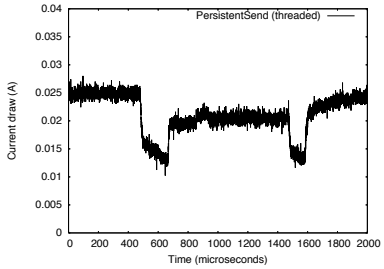
These results are consistent with our description of the TinyThread scheduler (Section 3.3). Recall that the entire TinyThread scheduler itself is a regular TinyOS task. It gets posted at startup time, and after that, it only gets posted when there is at least one *ready* thread waiting to execute. If all the threads in the program are inactive, the TinyThread scheduler never gets posted, and therefore does not cause any extra power drain. This is the key insight behind why the power utilization of a program is the same, regardless of whether TinyThread is used or not.

The second set of experiments related to power utilization we ran were to test the effect of using the radio. We wrote a simple application (called PersistentSend) that sent out a message on the radio every 30 milliseconds. The remaining time, the mote had its radio on in receive mode. Figures 9(a) and 9(b) shows the power utilization of this application running on a Tmote Sky. The radio's power is at its default level. At this level, the radio on this mote, the Chipcon CC2420 is supposed to consume 19.4 mA of current in the receive mode, and 17.4 mA of current while transmitting [6, 21]. This is consistent with our results. In the plot in Figure 9(a), we measure the power consumption of the unthreaded version of PersistentSend. In the left end of the PersistentSend plot, the mote has the radio turned on in receive mode. About 400μs into the reading, the mote switches the radio from receive mode to transmit mode, and then begins to transmit. The transmission lasts for about 800μs, after which the radio is switched back to the receive mode. The plot matches well with the values listed in the radio's data sheet. Figure 9(b), which is the threaded implementation of PersistentSend, displays a power profile that is identical to that of the regular TinyOS implementation.

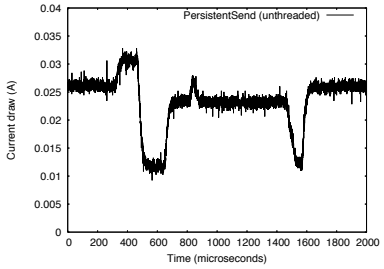
Figures 9(c) and 9(d) show the power profiles of the same PersistentSend application running on a micaz mote. The power profiles in this case are similar, although not identi-



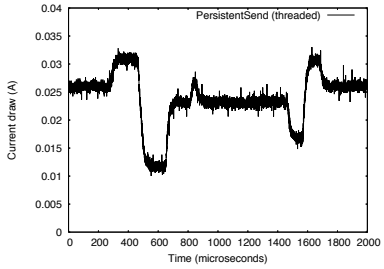
(a) Power draw of a Tmote Sky running PersistentSend (unthreaded).



(b) Power draw of a Tmote Sky running PersistentSend (threaded).



(c) Power draw of a micaZ running PersistentSend (unthreaded).



(d) Power draw of a micaZ running PersistentSend (threaded).

Figure 9. Comparison of power draw on a Tmote Sky running the unthreaded and threaded versions of the PersistentSend **application. This application keeps sending out a message on the radio every 30 milliseconds.**

cal. The extra power consumed is the processor performing the stack swap operations, and getting ready to execute the TinyThread scheduler. Just before the radio is switched to the Tx mode, the figure for the threaded application shows roughly an additional 50 μs duration where the current draw is at 30 mA. The same duration of increased power usage is seen at the other end (once the radio is done sending, and is being switched back to the Rx mode).

5.5 Response Time

Our second set of experiments measured how responsive programs were when implemented using TinyThread. These experiments study the effect of the overhead caused by the TinyThread scheduler. Analytically, this overhead is the number of instructions that the processor has to execute in order to perform the stack swapping. Every time the TinyThread scheduler is selected to execute, the stack swap is performed. Again, when the scheduler is done, and yields its spot on the processor, it has to swap the thread stack out, and replace the system stack in its original state.

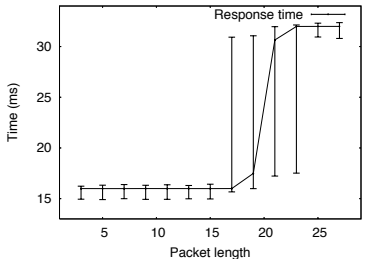
The first experiment we ran was a simple application (SimpleUART) on a mote that listened for messages coming from a PC over UART. When a message does arrive, it simply sends the same message back to the PC over UART. On the PC, a simple Java application sent 100 messages and averaged out the total round-trip time for each message. The time measurements on the PC were made using the actual processor clock via the Java Native Interface.

Figure 10(a) and 10(b) show the response times measured over a range of message sizes (3 bytes – 27 bytes). We show the median response time, along with the deviation over the 100 samples. The response times for the threaded version are similar to those of the unthreaded version, but it is possible to see the overhead caused by the thread library.

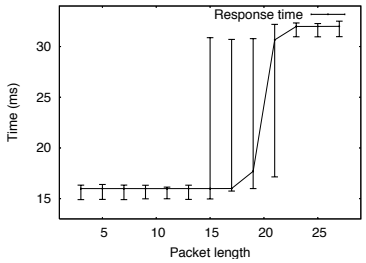
In order to study the effect of the second source of overhead, we designed an experiment in which there would always be two threads active at any time. This experiment computes a simple reconfigurable digital filter that of the kind normally seen in signal processing applications. Consider a light sensor that is sitting out in the open measuring the amount of UV radiation in sunlight. During the course of a day, there is a lot of noise that this sensor may see, *e.g.,* people may walk across, causing shadows. Rather than this sensor sending all the data points, including all the noise, it would be nice if the sensor could restrict the number of data points it sends to the collecting base station. However, the choice of which data points are safe to throw away is not trivial; this is where the digital filter helps.

A long-running calculation of this sort, however, does not fit in the traditional sensornet development model. The recommended way to perform such an operation in TinyOS is to break the calculation up into small tasks by ripping the stack manually. Using TinyThread however, the filter can be coded up very simply as a simple loop that performs some small subset of the calculation in each time slice. This thread therefore is always *ready*, and never terminates. To study response time in the presence of multiple *ready* threads, we superpose the SimpleUART application on top of this filter calculation, and measure the response time of the UART messages just like in the previous experiment.

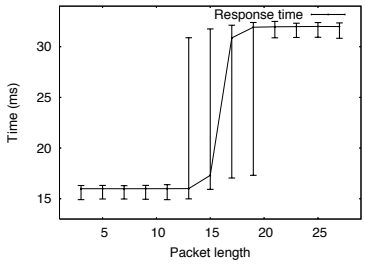
The results from this experiment are shown in Figures 10(c) and 10(d). In this case, the degradation in response time in the threaded version of the program is more percep-



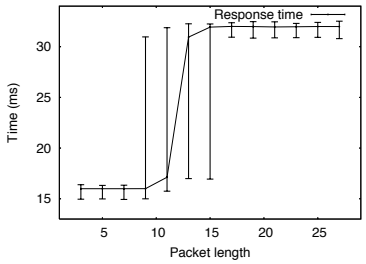
(a) Response time of a Tmote Sky running SimpleUART (unthreaded).



(b) Response time of a Tmote Sky running SimpleUART (threaded).



(c) Response time of a Tmote Sky running Filter, while receiving and responding to messages from PC (unthreaded).



(d) Response time of a Tmote Sky running Filter, while receiving and responding to messages from PC (threaded).

Figure 10. Comparison of round-trip message times from a PC to a mote and back.

tible. Notice how the response time jumps to the 30ms level when the packet size is 15 bytes in the threaded version, as opposed to 19 bytes in the event-driven version.

There is another source of reduced response as well. Suppose that there were two *ready* threads in a program. When the TinyThread scheduler is posted and executes, only one of the two threads gets a chance to execute. The other thread has to wait until TinyOS allows the TinyThread scheduler for the next time. This next chance may come after several other tasks and events in the program. By contrast, if the two threads were tasks, they would get a chance to execute in every "round" of the TinyOS task scheduler.

6 Discussion

6.1 Why Bother With Events Any More?

The question of whether to use events or threads to write highly-concurrent programs has been asked by several people before outside the sensornet space [1, 15, 27, 28]. Lauer and Needham have argued that the approaches are duals of each other [16]. We take a shot at it as well.

Now that we have a nice way to write energy-efficient multi-threaded programs for sensornets using TinyThread, can we totally get rid of event-based programming? Can we not follow the approach that MANTIS advocates, and write all of our programs in a strictly multi-threaded manner?

We do not subscribe to this view, for several reasons. As we pointed out in the introduction, event-based programming is well-suited when all the different parts of a program are functionally separate, and there is no state to be transferred among them. More concretely, as long as ripping a function into two parts does not transfer variables that originally belong on the stack to the global data section, the ripping does not cause any actual problems. The primary case where ripping a function into different tasks and event is a problem is when there is a data flow dependency. For example, Listing 8 shows the nesC code to send out a series of messages one to each neighbor in a graph.

Listing 8. Using a task and an event to send a number of messages. Notice how the program has to manually manage which message needs to be sent out next.

```
1  void sendAllMessages() {
2    lastMsgSent = 0;
3    post sendMessage(neighbors[lastMsgSent], msg);
4  }
5
6  task void sendMessage(uint16_t dest, TOS_MsgPtr msg)
7  {  call SendMsg.send(dest, sizeof(Message), &msg);  }
8
9  event result_t SendMsg.sendDone(TOS_MsgPtr sent,
10                                  result_t success) {
11   if (success) { // Message sent successfully; send next
12     lastMsgSent++;
13     if (lastMsgSent < numNeighbors)  // More neighbors
14       post sendMessage(neighbors[lastMsgSent], msg);
15   }
16   else // Message not sent successfully, resend
17     post sendMessage(neighbors[lastMsgSent], msg);
18 }
```

Notice how the program has to "remember" what it did last every time the `SendMsg.sendDone()` event is raised. This is an example of manual stack management. In addition to being cumbersome, it is also detrimental understanding the control flow logic. Using blocking I/O support in

Listing 9. Implementing a simple message forwarder. This is best implemented using an event, since there are no state dependencies.

```
event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr data) {
  Message *message = (Message *) data->data;
  call SendMsg.send(parent, sizeof(Message), &msg);
  return data;
}
```

TinyThread, the same effect can be achieved by:

```
for (i = 0; i < numNeighbors; i++)
  call Socket.send(neighbors[i], sizeof(Message), &msg);
```

When there are no data flow dependencies, however, events do offer an attractive way to achieve concurrent execution without having to deal with synchronization issues. Our approach with TinyThread is to approach what Adya *et al.* call a *"sweet spot"* in the design space of concurrent programs [1]. In this spirit, we allow events and threads to coexist. In our view, the correct way to use TinyThread is as a high-level programming abstraction. If some action in a program justifies an event, then it should be implemented as an event. For example, if a node is acting as a "dumb" message forwarder — it simply sends every message it receives to a known `parent` node, that action is better implemented in an event than in a thread (Listing 9).

This coexistence of events and threads has been an important design goal for TinyThread. This goal motivated us to extend an event-based language to add multi-threading rather than develop a new, purely multi-threaded language.

6.2 Preemptive vs. Cooperative Threading

There is a fundamental difference between threads in TinyThread and threads in say, modern operating systems such as Windows or Linux. The difference lies how threads are scheduled. In the latter case, scheduling is based on *time slices*. The scheduler simply allocates a time slice to a thread during which the thread executes. When the time slice is over, the executing thread is forced out of the processor immediately; it is preempted. These threading models use *preemptive* scheduling. In the case of TinyThread, however, the scheduling is done in a *cooperative* manner (akin to POSIX threads [5]). Each thread either runs to completion, or to some point in its execution where it has to block (typically an I/O operation, or a synchronization lock).

The distinction between preemptive and cooperative threads is similar to the distinction between native threads and green threads in Java [19]. Green threads are scheduled in a cooperative manner. When a high-priority green thread is executing, it will not be pre-empted. The only way another thread can get scheduled is if the executing thread yields or makes a blocking call, similar to the way our threads work. Native threads, on the other hand support true preemption.

This design decision of going with cooperative scheduling was a conscious one. When we set out to extend the nesC language with a thread library, our purpose was to solve problems, not add new ones. Preemptive multi-threading brings along with it a host of concurrency issues that need to be dealt with. nesC has a very clean solution to concurrent data access — if the compiler suspects that a data race could occur, a warning is given to the developer at compile-time. With preemptive multi-threading, such static analysis of data races is no longer possible. The compiler cannot predict when during its execution a thread T_1 may get preempted by another thread T_2. Consequently, if T_1 and T_2 share state, nesC cannot deterministically detect any possible data races.

In order to stay within the confines of the nesC data race detection model, TinyThread uses cooperative scheduling by default. However, the library also support true preemptive multi-threading. If a particular application really needed preemption, then TinyThread will support it, but without nesC's nice support for detecting possible data races.

6.3 TinyThread and nesC Data Race Detection

TinyThread allows the nesC data race detection to operate properly given a few caveats. The first major source of problem is the blocking calls. nesC does not have any understanding of blocking calls. This can create problems if users use these blocking calls in routines other than threads. TinyThread does provide some facilities for run-time avoidance of some of these faults, but there is no detection at compile time. The second caveat can occur when a developer makes a blocking call inside of an atomic statement. These problems can easily be overcome if blocking calls were added to the language specification and is an opportunity for future research.

7 Future Research

There are several different approaches which can significantly improve RAM consumption in threaded systems. One approach is to allow interrupts to execute on their own stack. This is implemented in the latest Linux kernel (commonly known as the *4K stack* configuration option), and in several other embedded operating systems.

As shown in Section 5, the overhead due to interrupts is required to be allocated for each thread. This forces threads to use significantly more memory than their event-driven counterparts. A separate interrupt stack can help in a more efficient implementation. Since an interrupt handler does not damage the current context, it does not actually need to swap the full context. The only register which must be swapped back and forth is the stack pointer. In an embedded system, this must be done with much care if interrupts can possibly nest, since there must be state denoting the stack in use.

Another possible direction for future work along similar lines is to implement a compiler which automatically generates event-driven code from cooperative threads. This problem of unspinning loops appears trivial, but when considering implementing this for several concurrent threads with several synchronization primitives and blocking I/O, it is far more complicated. This type of research in comparison with TinyThread could possibly reveal more insight into the strengths and weaknesses of both approaches.

TinyThread has the potential to push future work farther open than can be speculated. Algorithms thought to be out of practical reach on sensornets can now be easily implemented. New algorithms are only part of the possible future work enabled by TinyThread; applications of sensornets can

be pushed to new bounds. By easing the barrier of entry for new developers, more ideas and research could be spurred.

8 Conclusion

In this paper, we have presented TinyThread, a library-based approach to multi-threaded programming on sensornets. TinyThread is a clean library extension to nesC. Using this library does not require any changes to the nesC compiler or to the regular TinyOS toolchain. The library is written entirely in nesC.

In addition, we have presented stack-estimator, a static analysis tool that can compute exact stack requirements for MSP430- and AVR-based processors. The tool, in the TinyThread context, provides a way for developers to allocate individual context stacks for threads in a multi-threaded application, without having to worry about over- or under-estimating stack requirements.

Multi-threaded programming on the mote allows developers to write programs using a procedural style: the style of programming that is taught to every CS student. Since TinyThread is fully integrated with the rest of nesC, it could also potentially act as a nice way for programmers conversant with say, Java, to "ease in" to nesC programming.

Based on our experimental evaluation, we report that the primary design metric for sensornets — power utilization — is virtually unaffected by TinyThread. The TinyThread scheduler is intelligent enough to schedule threads only when active. The rest of the time, the scheduler is sleeping, allowing the mote to conserve power.

9 References

[1] A.Adya *et al*. Cooperative task management without manual stack management. In *USENIX Annual Technical Conference, General Track*, pages 289–302, 2002.

[2] Atmel Corporation. Atmega128(l) data sheet. www.atmel.-com/dyn/resources/prod_documents/doc2467.pdf.

[3] S. Bhatti *et al*. MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms. *Mobile Networks and Applications*, 10(4):563–579, aug 2005.

[4] D. Brylow, N. Damgaard, and J. Palsberg. Static checking of interrupt-driven software. In *ICSE01*, 2001.

[5] D. R. Butenhof. *Programming with POSIX threads*. Addison-Wesley, Boston, MA, USA, 1997.

[6] Chipcon, Texas Instruments. Cc2420 data sheet. www.chipcon.com/files/CC2420_Data_Sheet_1_3.pdf.

[7] A. Dunkels, O. Schmidt, and T. Voigt. Using Protothreads for Sensor Node Programming. In *REALWSN'05*, Stockholm, Sweden, June 2005.

[8] Dunkels A. *et al.* Contiki - a lightweight and flexible operating system for tiny networked sensors. In *LCN '04*, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.

[9] R. S. Engelschall. Portable multithreading: The signal stack trick for user-space thread creation. In *2000 USENIX Annual Technical Conference*, 2000.

[10] A. K. *et al.* Parallel programming in split-c. In *Supercomputing '93*, pages 262–273, New York, NY, USA, 1993.

[11] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. In *DCOSS '05*, June 2005.

[12] Hill J. *et al.* System architecture directions for networked sensors. In *ASPLOS-IX*, pages 93–104, New York, NY, USA, 2000. ACM Press.

[13] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04*, pages 81–94. ACM Press, 2004.

[14] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Trans. Netw.*, 11(1):2–16, 2003.

[15] O. Kasten and K. Römer. Beyond event handlers: Programming wireless sensors with attributed state machines. In *IPSN '05*, pages 45–52, Los Angeles, USA, Apr. 2005.

[16] H. C. Lauer and R. M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, 1979.

[17] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. In *ASPLOS-X*, pages 85–95, New York, NY, USA, 2002. ACM Press.

[18] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *NSDI '04*, 2004.

[19] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley, Boston, MA, USA, 1999.

[20] T. D. May, S. H. Dunning, G. A. Dowding, and J. O. Hallstrom. An rpc design for wireless sensor networks. *J. Pervasive Computing and Communication*, March 2006.

[21] Moteiv Corporation. Tmote Sky data sheet. moteiv.com/products/docs/tmote-sky-datasheet.pdf.

[22] R. Newton and M. Welsh. Region streams: functional macro-programming for sensor networks. In *DMSN '04*, pages 78–87, New York, NY, USA, 2004. ACM Press.

[23] C. Nitta, R. Pandey, and Y. Ramin. Y-Threads: Supporting concurrency in wireless sensor networks. In *LNCS 4026 (DCOSS '06)*, pages 169–184, jun 2006.

[24] J. Regehr and A. Reid. Hoist: a system for automatically deriving static analyzers for embedded systems. In *ASPLOS-XI*, pages 133–143, New York, NY, USA, 2004. ACM Press.

[25] J. Regehr, A. Reid, and K. Webb. Eliminating stack overflow by abstract interpretation. *Trans. on Embedded Computing Sys.*, 4(4):751–778, 2005.

[26] Texas Instruments. MSP430x1xx family user's guide (rev. f). www-s.ti.com/sc/psheets/slau049f/slau049f.pdf.

[27] J. R. von Behren, J. Condit, and E. A. Brewer. Why events are a bad idea (for high-concurrency servers). In M. B. Jones, editor, *HotOS*, pages 19–24. USENIX, 2003.

[28] J. R. von Behren *et al.* Capriccio: scalable threads for internet services. In *SOSP '03*, pages 268–281, New York, NY, USA, 2003. ACM Press.

[29] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *NSDI*, pages 29–42. USENIX, 2004.

[30] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *MobiSys '04*, pages 99–110, New York, NY, USA, 2004. ACM Press.