TOSThreads: Thread-safe and Non-Invasive Preemption in TinyOS

Kevin Klues[‡], Chieh-Jan Mike Liang[†], Jeongyeup Paek[○], Răzvan Musăloiu-E.[†], Philip Levis[⋆], Andreas Terzis[†], Ramesh Govindan[○]

[‡] UC Berkeley, Berkeley, CA
[†] Johns Hopkins University, Baltimore, MD
[○] University of Southern California, Los Angeles, CA
[⋆] Stanford University, Stanford, CA

klueska@cs.berkeley.edu {cliang4, razvanm, terzis}@cs.jhu.edu {jpaek, ramesh}@usc.edu pal@cs.stanford.edu

Abstract

Many threads packages have been proposed for programming wireless sensor platforms. However, many sensor network operating systems still choose to provide an event-driven model, due to efficiency concerns. We present TOSThreads, a threads package for TinyOS that combines the ease of a threaded programming model with the efficiency of an event-based kernel. TOSThreads is backwards compatible with existing TinyOS code, supports an evolvable, thread-safe kernel API, and enables flexible application development through dynamic linking and loading. In TOSThreads, TinyOS code runs at a higher priority than application threads and all kernel operations are invoked only via message passing, never directly, ensuring thread-safety while enabling maximal concurrency. The TOSThreads package is non-invasive; it does not require any large-scale changes to existing TinyOS code.

We demonstrate that TOSThreads context switches and system calls introduce an overhead of less than 0.92% and that dynamic linking and loading takes as little as 90 ms for a representative sensing application. We compare different programming models built using TOSThreads, including standard C with blocking system calls and a reimplementation of Tenet. Additionally, we demonstrate that TOSThreads is able to run computationally intensive tasks without adversely affecting the timing of critical OS services.

Categories and Subject Descriptors

D.4 [**Operating Systems**]: Organization and Design; D.1.3 [**Software**]: Programming Techniques

General Terms

Design, Performance

Keywords

TinyOS, Multi-Threading, Sensor Networks

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SenSys'09, November 4–6, 2009, Berkeley, CA, USA.
Copyright 2009 ACM 978-1-60558-748-6 ...$5.00

1 Introduction

Many mote operating systems use event-driven execution to support multiple concurrent execution contexts with the memory cost of a single stack [9, 15, 18]. Network protocols, storage subsystems, and simple data filters can be easily developed in this model, as they typically perform short computations in response to I/O events. More generally, there are sound reasons for mote OSs to be event-based: given the motes' memory and processing constraints, an event-based OS permits greater concurrency than other alternatives.

The event-driven programming model is less well-suited for developing higher-level services and applications. In this model, application developers need to explicitly manage yield points or continuations, or partition a long-running computation to avoid missing events. Compression is a concrete example of a long-running computation that can plausibly be implemented on advanced sensor platforms (such as the imote2 [6]). Many sensor network applications, such as seismic sensing [37], structural monitoring [5, 20], and image-based sensing [17], could benefit greatly from data compression. Indeed, Sadler et al. showed that compression can reduce energy consumption significantly [33]. Nevertheless, real sensor network deployments rarely use it due to the difficulty of implementing it in event-driven environments.

In this paper, we explore the tension between achieving high concurrency in the face of resource constraints and having an intuitive programming model for long-running computations. While thread-based programming models have been successful in many embedded OSs such as uC/OS [28], FreeRTOS [1], and many others, prior attempts to use them in the sensor network domain have had limited success.

Cooperative threading approaches, exemplified by TinyThreads [27], rely on applications to explicitly yield the processor, thereby placing the burden of managing concurrency explicitly on the programmer. As we show in Section 2, this task can be quite difficult for long-running computations such as compression. These computations are data-dependent, so the commonly-used strategy of placing fixed yield points in the code can result in highly-variable inter-yield intervals. To ensure non-invasive behavior with timing sensitive OS services, the programmer must design sophisticated and burdensome strategies that explicitly monitor the execution time of the application.

Preemptive threading approaches, exemplified by TinyMOS [35] and Contiki's [9] optional preemptive threading library, have had some limitations as well. TinyMOS [35], an extension of the thread-based Mantis OS [2], runs TinyOS inside a dedicated Mantis thread, but requires placing a giant lock around most of the TinyOS code. This approach limits overall concurrency and efficiency, as the TinyOS thread is only able to service one application thread at a time. Contiki [9], an event-based operating system similar to TinyOS, provides hooks for implementing preemptive threading on top of its event-based kernel, but provides limited mechanisms to control re-entrancy when invoking shared OS services. As we discuss in Section 2, the lack of standardized mechanisms for synchronizing thread-based application processing with event-based kernel processing can lead to some subtle race conditions distributed across a large body of code.

We present TOSThreads, a fully preemptive threads package for TinyOS. TOSThreads resolves the tension between the ease of thread-based programming and the efficiency of event-based programming by running all event-based code inside a single high priority *kernel thread* and all application code inside *application threads*, which only execute whenever the kernel thread becomes idle. Application threads invoke kernel operations via message passing, so that application threads never run kernel code themselves. This architecture ensures that all TOSThreads operations that invoke the kernel are both *thread-safe* (i.e., thread preemption cannot cause the kernel to fail) and *non-invasive* with respect to timing-sensitive kernel operations (i.e., thread priorities are always preserved).

While each of the techniques adopted by TOSThreads to achieve these goals are not novel themselves, TOSThreads is the first threads package to combine them in a way that makes preemptive threading a viable option for sensor networks. For example, while message passing has been widely studied in the context of microkernel architectures, the overhead induced by virtual memory has restricted widespread adoption of this technique in general-purpose OSs. TOSThreads revives this technique since low-power microcontrollers do not suffer the major costs of context switches seen by high-performance processors (virtual memory, TLB flushes, pipelined execution).

In addition to thread-safety and non-invasiveness, the design and implementation of TOSThreads is influenced by three secondary goals chosen to encourage widespread adoption: backwards compatibility with TinyOS, an easily extensible kernel API, and support for flexible application development. To our knowledge, prior work on threading libraries for sensor networks has not attempted to collectively achieve each of these goals.

As the de facto standard development platform for many sensor network applications, TinyOS has been ported to a wide variety of platforms and contains many subsystems that are in wide use today (e.g. routing, time synchronization, dissemination, collection, etc.). In order to preserve backwards compatibility with these subsystems, TOSThreads must preserve all of the efficiency and timing constraints imposed by them. TOSThreads achieves this goal by enforcing a strict priority between event-based TinyOS code and application threads. TinyOS code is guaranteed to take precedence over application threads, and application threads simply run as background tasks when the TinyOS thread has nothing more to do.

TOSThreads second goal of extensibility ensures that developers are able to easily integrate new services into the TOSThreads kernel API as more and more subsystems continue to be developed for TinyOS. TOSThreads achieves this goal by providing a well defined API of blocking system calls based on message passing. System developers can easily evolve this API by wrapping additional blocking system calls around any new event-driven TinyOS service. Moreover, by exporting this API in both nesC [13] and ANSI C, we allow developers to implement efficient applications without the need to learn a new language.

TOSThreads final goal of support for flexible application development enables programmers to develop their applications using multiple programming paradigms and provide a way to dynamically link and load executable objects received over the network. To substantiate this claim we have reimplemented the Tenet programming system [14], on top of TOSThreads. The resulting system has higher expressivity without increasing code size. Additionally, TOSThreads has enabled the development of a novel programming language, *Latte*, a JavaScript dialect that compiles to C.

As we show in Section 5, TOSThreads is able to achieve each of its goals with minimal overhead in terms of execution speed and energy consumption, while still efficiently supporting time-critical OS services in the presence of long-running background computations.

The rest of this paper is organized as follows. Section 2 provides some background on the general challenges of implementing a preemptive threading library on top of an existing event-based kernel. Section 3 introduces the basic architecture of our TOSThreads preemptive threads package. Section 4 provides an overview of the TOSThreads implementation in TinyOS. Section 5 discusses some evaluation results. Section 6 presents some related work, and Section 7 concludes.

2 The Challenge of Preemption

Fully preemptive threads raise a number of difficult implementation challenges. The most far-reaching of these is empowering fully preemptive application code with the ability to call kernel code. If the kernel scheduler preempts a thread in the midst of running kernel code, it is possible that the newly scheduled thread could end up making a system call which will invoke kernel code as well. For correctness of program execution, using such a preemption model requires all kernel code to be fully reentrant, allowing any number of threads to concurrently access the kernel at any time.

Making all kernel code fully preemptive is a daunting task. There are three common techniques to attack the problem, of which we discuss two (cooperative threading and kernel locking) in this section. TOSThreads uses a third approach, message passing, discussed in the next section.

2.1 Cooperative threading

Cooperative threading avoids the challenge of kernel reentrancy. Instead, the kernel context switches between thr-

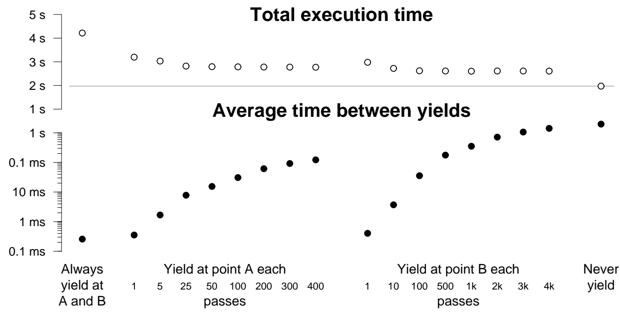

Figure 1. Total execution time and average time between yields of various yielding strategies for a compression algorithm applied on a set of 13 gray-scale images captured with the Cyclops platform.

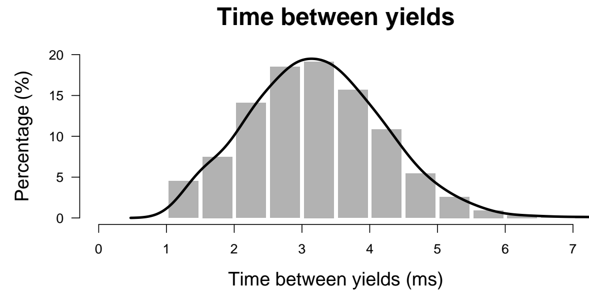

Figure 2. Distribution of the time between yields for yielding every ten passes over point B for the set of images used in Figure 1.

eads only on a set of well-known system calls, such as I/O operations. Kernels also typically have a `yield()` system call, which does nothing except explicitly yielding the processor to other, runnable threads. Among threads packages proposed for sensor systems, TinyThreads takes this approach [27]. In TinyThreads, TinyOS tasks run in a dedicated system thread and application threads cooperatively share the processor with this system thread.

The challenge with cooperative threading is that the correctness of the entire system depends upon application code voluntarily yielding the processor at specific intervals. If a thread does not relinquish the processor for a long time, then other threads may be unable to service requests, read sensors, or otherwise perform their normal tasks in a timely manner. A long-running application thread can greatly delay TinyOS tasks from running, in effect violating the TinyOS programming mantra of "keep tasks short".

Furthermore, determining the right strategy for inserting yield points in a long running computation is a non-trivial exercise. As an example, we examine how yield points affect a simple image compression algorithm one might implement on a sensor node with a camera. The algorithm has a doubly-nested loop. The outer-loop iterates over the rows of the image, while the inner loop compresses a single row using run-length encoding. A similar compression algorithm was used in the deployment described in [17].

Figure 1 shows the effect of adding an explicit yield to the inner (point A) or the outer loop (point B) for a set of 13 images from the deployment in [17]. Simply adding a yield to each execution of both loops introduces tremendous overhead: the computation takes 213% longer. However, naively inserting a yield after every ten executions of the inner loop is no better either; assuming an inter-packet arrival interval of 3ms, its possible that yielding this infrequently may cause a node to start to drop packets. In fact, considering the distribution of inter-yield times shown in Figure 2, one can see that more than half of the algorithm's iterations would take longer than the desired 3 ms. Moreover, the inter-yield times actually depend on the compressibility of each image, so we cannot even generalize the results presented here to a different set of images using the same compression algorithm.

Preemptive threading relieves the programmer of the burden of determining when to explicitly yield the processor. Modern operating systems all incorporate preemptive threading because the costs and dangers of cooperative threading greatly outweigh its benefits. Making the kernel slightly more complex with support for preemptive threading simplifies all application code.

2.2 Kernel locking

The second major approach is to put mutual exclusion locks (mutexes) within the kernel. In the simplest variant, the entire kernel has a single lock around it, such that only one thread can be executing a system call at any time. This is the approach taken in TinyMOS [35] as well as the approach taken in early versions of the Linux kernel.

More fine-grained locking is also possible: there is a basic trade-off between the locking overhead and how much parallelism the system can support. TinyMOS, for example, also points out that each subsystem (radio, sensors, flash) can have its own lock, so that one thread could access storage while another sends packets. In the extreme case, locks can be very fine-grained (e.g., per data structure).

Unlike cooperative threading, a malicious or buggy thread in a kernel locking system cannot monopolize the processor. The cost is that the kernel is more complex, and in the absence of parallelism the locks reduce performance. Modern multitasking OSes (Linux, Solaris, Windows, the BSDs) all use kernel locking, with differing lock granularities.

2.3 The Subtleties of Architecting Preemption

Before we discuss our implementation of TOSThreads, it is useful to present a case study of our experience implementing a preemptive threading library for another event-based sensor network OS, Contiki. Given the similarities between Contiki and TinyOS, this case study serves as a starting point for discussing the subtleties involved in architecting a preemptive threading library on top of an event-based kernel.

The Contiki source code provides documentation that outlines the process of extending its native threading library to support preemptible threads. Adding this support is as simple as extending a timer interrupt handler to call its thread scheduling function whenever it fires. While this simple extension does enable preemptive threading support for applications, it comes with some limitations.

The problem is that the Contiki kernel provides limited mechanisms to control reentrancy when invoking shared OS services. (i.e., the API exposed to applications is not guaranteed to be thread-safe). Application threads must explicitly synchronize on every kernel access, complicating application code and distributing the source of potential race conditions across a large number of functions written by a disparate set of developers. As we discuss in the following section, TOSThreads avoids this problem by extending the TinyOS concurrency model to explicitly support threads and provides a message passing interface to automatically enforce thread-safety when invoking shared OS services. While Contiki can inherently provide similar functionality, there is no support for it in its current implementation.

To test what would happen if we actually launched a set of application threads that invoked a shared OS service without explicit synchronization, we implemented preemptive threading in Contiki according to the techniques outlined in its source code. We then ran three threads that executed tight I/O loops. In each execution of the loop, a thread allocated a memory buffer through Contiki's MEMB abstraction and sent it over the radio; calls to the MEMB API were not explicitly synchronized and access to the radio was not arbitrated by the application. We introduced two checks to monitor the behavior of the system while these three threads were running. In the first, we checked whether the memory allocator ever returned the same pointer to two different threads: this would be caused by unsynchronized access to the allocator function while modifying the memory data structure. In the second, we instrumented the lock used by Contiki's CC2420 stack to check if critical sections were ever entered by more than one thread at a time.

In both experiments, it took under a second for one of these two checks to be violated. This speed is, of course, somewhat artificial, due to the execution of three tight loops (i.e., the occurrence of the fault would take much longer in a low-power, low-duty cycle application). Nevertheless, these experiments demonstrate (unsurprisingly), that proper synchronization is critical for correct operation of the system. Even with some level of synchronization, care must be taken to ensure that concurrency is not prohibitively limited.

In summary, preemptive threads are a non-trivial mechanism for an operating system to correctly support. Prior approaches, such as TinyMOS [35] and TinyThreads [27], limit preemption by either discarding it completely (TinyThreads) or limiting it to application code alone (TinyMOS). Finally, while event-driven systems such as Contiki and TinyOS are well-equipped to support preemptive threading, one must take special care to avoid limiting the concurrency of the overall system. In the following section we discuss the message passing approach taken by TOSThreads which enables full preemption with maximal concurrency.

3 TOSThreads Architecture

This section describes the TOSThreads architecture. It provides an overview of the basic functionality provided by TOSThreads, a description of its overall structure, a list of modifications that had to be made to TinyOS to enable TOSThreads, a description of the flexible user/kernel boundary defined by TOSThreads, and a short description of the mechanisms available for performing dynamic linking and loading of TOSThreads binaries.

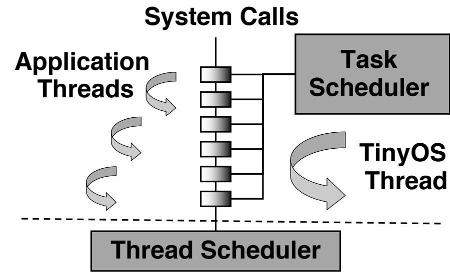

Figure 3. Overview of the basic TOSThreads architecture. The vertical line separates user-level code on the left from kernel code on the right. Looping arrows indicate running threads, while the blocks in the middle of the figure indicate API slots for making a system call.

3.1 Overview

The existing TinyOS concurrency model has two classes of execution: synchronous (tasks) and asynchronous (interrupts). These two classes of execution follow a strict hierarchy whereby asynchronous code can preempt synchronous code but synchronous code is run-to-completion. TOSThreads extends this concurrency model to provide a third execution class in the form of user-level application threads. Application threads exist at the lowest level of the hierarchy and are prohibited from preempting either synchronous code or asynchronous code (but are still allowed to preempt one another). Application threads synchronize using standard primitives such as mutexes, semaphores, barriers, and condition variables.

TOSThreads exploit message passing by running TinyOS inside a special, high priority kernel thread dedicated to running the standard TinyOS task scheduler. When an application thread makes a system call, it does not directly call into or execute TinyOS code itself. Instead, it passes a message to the kernel thread by posting a TinyOS task. Since posting a task causes the high priority kernel thread to wake up, it will immediately preempt the active application thread and begin executing the system call. As TinyOS is completely non-blocking, this execution is fast and the kernel thread quickly returns to sleep. At this point, control returns to the thread scheduler, allowing application threads to resume.

By using message passing, TOSThreads ensures that only a single thread of control – the kernel thread – ever directly executes TinyOS code. This allows core TinyOS code to execute *unchanged* and requires no additional synchronization primitives on either side of the message passing interface. Furthermore, TOSThreads does not constrain concurrency within TinyOS itself, as it does not introduce any limitations on what system calls can be made. While TOSThreads cannot remove the self-imposed concurrency limits of a TinyOS abstraction (e.g., can only handle one outstanding operation), it does not add any further limitations.

3.2 Structure

Figure 3 presents the overall structure of a TOSThreads system, consisting of five key elements: the TinyOS task scheduler, a single kernel-level TinyOS thread[1], a thread scheduler, a set of application threads, and a set of system call APIs and their corresponding implementations. Any number of application threads can exist concurrently (barring memory constraints) and a single kernel thread is used to run the TinyOS task scheduler. The underlying thread scheduler manages the concurrency between all application threads and the system call APIs provide an abstraction on top of the underlying message passing interface used to communicate with the TinyOS kernel.

There are two ways in which posted events can cause the TinyOS thread to wake up. First, an application thread can issue a blocking system call into the TinyOS kernel. This call internally posts a task, implicitly waking up the TinyOS thread to process it. Second, an interrupt handler can post a task for deferred computation. Since interrupt handlers run on the stack of the currently active thread, if an interrupt handler posts a task, TOSThreads must context switch to the TinyOS thread. Control eventually returns to the application thread after the TinyOS thread has emptied the task queue.

3.3 TinyOS Modifications

Only two changes to the existing TinyOS code base are required to support TOSThreads: a modification to the boot sequence and the addition of a post-amble for every interrupt handler. The change in the boot sequence encapsulates TinyOS inside the single kernel-level thread before it boots. Once it runs, TinyOS operates as usual, passing control to the thread scheduler at the point when it would have otherwise put the processor to sleep. The interrupt handler post-ambles ensure that TinyOS runs when an interrupt handler posts a task to its task queue. As discussed in Section 5, our evaluations show these modifications only decrease the performance of TinyOS by a negligible amount.

3.4 Flexible User/Kernel Boundary

One significant difference between TOSThreads and other TinyOS threading implementations is that TOSThreads defines a flexible boundary between user code and kernel code. Rather than dividing code into user and kernel space based on access rights to privileged operations, TOSThreads loosely defines a conceptual user/kernel boundary as the point in which programs switch from a threaded to an event-driven programming model. Because all existing TinyOS code is event-driven, any component in the current TinyOS code base can be included in a TOSThreads kernel.

TOSThreads makes building a kernel from existing TinyOS components a straightforward process. Just as a traditional TinyOS application consists of the TinyOS task scheduler and a custom graph of components, a TOSThreads kernel consists of the task scheduler, a custom graph of components, and a custom set of blocking system calls. Each of these calls is a thin wrapper on top of an existing TinyOS service (e.g., active messaging, sensing, multi-hop routing). The wrapper's sole purpose is to convert the non-blocking *split-phase* operation of the underlying TinyOS service into a blocking one based on message passing. The API that a kernel ultimately provides depends on the set of TinyOS services its designer wishes to present to applications.

Through this flexible user/kernel boundary, TOSThreads enables the kernel to evolve in support of diverse user-level code bases. We demonstrate this ability by developing two custom TOSThreads kernels: one that provides a standard set of TinyOS services (Section 4.3) and one that implements the Tenet API (Section 5.6).

3.5 Linking and Loading

Defining an explicit user/kernel boundary creates the possibility of compiling applications separately and dynamically linking them to a static kernel. TinyLD is the TOSThreads component implemented to provide this functionality. To use TinyLD, users write a standalone application that invokes system calls in the kernel API. This application is compiled into an object file and compressed into a custom *MicroExe* format we have developed. The compressed binary is then transported to a mote using standard methods (e.g., serial interface, over-the-air dissemination protocol, etc.). Once on the mote, TinyLD dynamically links the binary to the TinyOS kernel, loads it into the mote's program memory and executes it.

4 Implementation

This section describes the implementation of TOSThreads, including the internals of the thread scheduler, the thread and system call data structures, and the dynamic linking and loading process. While most of the TOSThreads code is platform independent, each supported platform must define platform-specific functions for (1) invoking assembly language instructions to perform a context switch and (2) adding a post-amble to every interrupt handler. Defining these functions is a fairly straightforward process, and support exists for Tmote Sky, Mica2, Mica2dot, MicaZ, Iris, eyesIFX, Shimmer, and TinyNode motes. As of TinyOS 2.1.0, TOSThreads is included as part of the standard TinyOS distribution and documentation on implementing applications that use it can be found on the TinyOS documentation wiki [34].

4.1 The Thread Scheduler

TOSThreads exposes a relatively standard API for creating and manipulating threads: `create()`, `destroy()`, `pause()`, `resume()` and `join()`. These functions form part of the system call API and can be invoked by any application program.

Internally, TOSThreads components use thread scheduler commands that allow them to `initialize()`, `start()`, `stop()`, `suspend()`, `interrupt()`, or `wakeup()` a specific thread. The thread scheduler itself does not exist in any particular execution context (i.e., it is not a thread and does not have its own stack). Instead, any TOSThreads component that invokes one of these commands, executes in the context of the calling thread; only the interrupt handler post-ambles and the system call API wrappers invoke them directly.

The default TOSThreads scheduler implements a fully preemptive round-robin scheduling policy with a time slice of 5 msec. We chose this value to achieve low latency across

[1] In what follows, we use the terms "TinyOS thread" and "kernel thread" interchangeably.

multiple application-level computing tasks. While application threads currently run with the same priority, one can modify the scheduler to support more sophisticated policies.

The thread scheduler is the first component to take control of the processor during the boot process. Its job is to encapsulate TinyOS inside a thread and trigger the normal TinyOS boot sequence. Once TinyOS boots and processes all of its initial tasks, control returns to the thread scheduler which begins scheduling application threads. The scheduler keeps threads ready for processing on a ready queue, while threads blocked on I/O requests or waiting on a lock are kept on different queues. Calling `interrupt()` or `suspend()` places a thread onto one of these queues while calling `wakeup()` removes it from a queue.

4.2 Threads

TOSThreads dynamically allocates Thread Control Blocks (TCB) with space for a fixed size stack that does not grow over time. While the memory costs associated with maintaining per thread stacks can be substantial, we believe the benefits of the programming model provided by preemptive threading outweigh these costs in many situations. That said, one can use techniques such as those proposed by McCartney and Sridhar [27] to estimate (and thereby minimize) the memory required by each of these stacks.

In order to aid our discussion of TOSThreads internals, we provide the following code snippet which shows the complete structure of a TOSThreads TCB. Below, we describe each of the included fields in more detail:

```
struct thread {
  thread_id_t thread_id;
  init_block_t* init_block;
  struct thread* next_thread;

  uint8_t mutex_count;      //
  uint8_t state;            // thread_state
  thread_regs_t regs;       //

  void (*start_ptr)(void*); // start_function
  void* start_arg_ptr;

  uint8_t joinedOnMe[];
  stack_ptr_t stack_ptr;
  syscall_t* syscall;
};
```

`thread_id`: This field stores a thread's unique identifier. It is used primarily by system call implementations and synchronization primitives to identify the thread that should be blocked or woken up.

`init_block`: Applications use this field when dynamically loaded onto a mote. As Section 4.4 describes, whenever the system dynamically loads a TOSThreads application, the threads it creates must receive all the state associated with its global variables. An initialization block structure stores these global variables and `init_block` points to it.

`next_thread`: TOSThreads uses a single queue mechanism to keep track of blocked threads and to maintain the ready queue. These queues are implemented as linked lists of threads connected through their `next_thread` pointers. By design, a single pointer suffices: threads are *always* added to a queue just before they are interrupted and are removed from a queue before they wake up. This method saves memory.

`thread_state`: This set of fields store information about a thread's current state. Specifically, it contains a count of the number of mutexes the thread currently holds; a state variable indicating the thread's state (INACTIVE, READY, SUSPENDED, or ACTIVE); and a set of variables that store the processor's register set when a context switch occurs.

`start_function`: This set of fields point to a thread's start function along with a pointer to a single argument. The application developer must ensure that the structure the argument points to is not deallocated before the thread's `start()` function executes. These semantics are similar to those defined by the Unix `pthreads` library.

`joinedOnMe`: This field stores a bitmap of the thread ids for all threads *joined* on the current thread through a `join()` system call. This bitmap is traversed when the current thread terminates and any threads waiting on it are woken up.

`stack_ptr`: This field points to the top of a thread's stack. Whenever a context switch is about to occur, the thread scheduler calls the `switch_threads()` function, pushing the return address onto the current thread's stack. This function stores the current thread's register state, replaces the processor's stack pointer with that of a new thread, and finally restores the register state of the new thread. Once this function returns, the new thread resumes its execution from the point it was interrupted.

`syscall`: This field contains a pointer to the message passed between an application thread and the TinyOS kernel when making a system call. Access to this structure is shared between these two threads, with the application thread writing the message into the structure and the kernel thread reading it out. The following section explains how this structure is used in more detail.

4.3 Blocking Syscall API

TOSThreads implements blocking system calls by wrapping existing TinyOS services inside blocking APIs. These wrappers are responsible for packing the parameters supplied to a system call into a message and posting a task to the TinyOS kernel. These wrappers also maintain state across the non-blocking *split-phase* operations associated with an underlying TinyOS service while it is processing a system call. All wrappers are written in nesC with an additional layer of C code layered on top of them. We refer to the TOSThreads *standard* C API as the API providing system calls to standard TinyOS services such as sending packets, sampling sensors, and writing to flash. Alternative APIs (potentially also written in C) can be implemented as well (e.g., the Tenet API discussed in Section 5.6). For more information on the full API provided by TOSThreads, please refer to the TOSThreads section of the TinyOS documentation wiki [34].

A user thread initiates a system call by calling a function provided by one of the blocking system call wrappers. Each message passed by a wrapper function to the TinyOS kernel contains a unique `syscall_id` associated with the system call, a pointer to the `thread` invoking the call, a pointer to

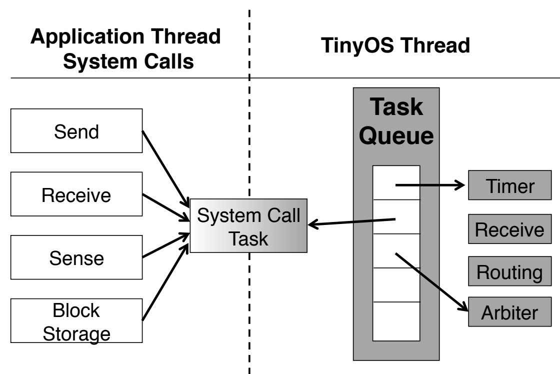

Figure 4. TOSThreads exposes kernel APIs through blocking system calls wrapped around event-driven TinyOS services. These wrappers (white boxes on the left) run their respective system calls inside a single shared TinyOS task. This task is interleaved with other TinyOS tasks (grey boxes on the right) which TinyOS itself posts.

the function that TinyOS should call once it assumes control, and the set of parameters this function should receive.

All variables associated with a system call message (i.e., the pointer to the SCB and the parameters passed to the system call itself) are allocated on the local stack of the application thread at the time of the system call. This is possible because once the calling thread invokes a system call, it will not return from the function which instantiates these variables until after the blocking system call completes. The message remains on the application thread's stack throughout the duration of the system call and can therefore be accessed by the kernel thread as necessary.

As discussed in Section 3, making a system call implicitly posts a TinyOS task, causing the TinyOS thread to immediately wake up and the calling thread to block. Because we constrain our design to allow the TinyOS thread to run whenever it has something to do, it is unnecessary to maintain a queue of system calls waiting to be initiated; the downcall of one system call will always complete before it is possible for another thread to initiate a system call. Thus, only one TinyOS task is necessary to perform all application system calls. The body of this task simply invokes the function pointed to in the message passed by the calling thread. This is in contrast to many other threads packages which must maintain system call queues to track the system calls that each thread makes. Figure 4 provides a visual representation of the TOSThreads approach.

4.4 Dynamic Linker and Loader

Once we provide a clear separation between user and kernel code and provide a system call API through which they can interact, we enable the possibility of compiling applications separately and loading them dynamically at runtime. TinyLD is the dynamic linker and loader we implemented for TOSThreads. Using TinyLD, application programs written in C can be dynamically installed and simultaneously executed on a mote. TinyLD's task is to resolve, at run-time, references to kernel API calls made by a dynamically loadable application. To enable this, an application is first compiled offline into a customized loadable binary format called MicroExe. This binary is then distributed to a mote via a dissemination protocol, or installed on a mote via its serial interface. At that point, TinyLD accesses the binary from flash or RAM, patches unresolved address references, and links it to the kernel. Finally, TinyLD loads the resulting binary to the mote's ROM and spawns a thread to execute it. Currently, the MicroExe format is only supported on MSP430-based platforms, but we are in the process of modifying it to support others as well. The paragraphs that follow elaborate on the MicroExe format used by the linking and loading process.

4.4.1 MicroExe

Binary formats for dynamic linking and loading on general purpose OSs are inefficient for memory-constrained mote platforms [9]. Consider the Executable and Linkable Format (ELF), the most widely used format for dynamic linking and loading in Unix systems. While ELF encodes addresses as 32-bit values, mote platforms based on the MSP430 microcontroller, such as the Tmote Sky [29], have a 16-bit address space. Moreover, symbol names in ELF are encoded as text strings for ease of use, thus increasing the size of the file.

MicroExe is a file format designed specifically for loadable binaries in TinyOS. It uses 16-bit addresses, compacts the representation of symbol names, and uses *chained references* techniques ([23]) to reduce its size. Although it is optimized for the MSP430 platform, its design principles are equally applicable to other micro-controllers. A MicroExe binary is created by first compiling an application written in C to an ELF binary using a standard GCC compiler. Next, a generator script running on a PC invokes the GCC toolchain to extract ELF symbol and relocation tables to construct a semantically equivalent, yet space-optimized, MicroExe file. A detailed description of the MicroExe file format can be found in the MicroExe technical report [30].

4.4.2 Linking and Loading

The linking and loading process consists of four steps. First, TinyLD links the binary's machine code to the kernel by patching unresolved addresses corresponding to calls for kernel services. It then allocates memory for all global variables that the binary defines, patches references to local variables, and loads the machine code into the mote's ROM. These steps are conceptually straightforward and all information required for them is encoded in the MicroExe file.

Once all of the linking and loading steps are complete, TinyLD invokes TOSThreads to spawn a new *root thread* and begins running the application binary. A pointer to `tosthread_main`, the starting point of the thread, as well as a pointer to all global variables associated with the application are passed as arguments to the new root thread, inside the special `init_block` structure described in Section 4.2. Once the newly spawned thread starts running, it calls `tosthread_main` and waits for the entire program to finish before terminating. The `init_block` structure remains active throughout the application's lifetime and can be referenced by any threads that `tosthread_main` or any of its children ultimately spawn.

Operation	Number of cycles
Start Static Thread	283
Start Dynamic Thread	679 + *malloc*()
Interrupt Thread	100
Suspend Thread	145
Wakeup Thread	15
Static Thread Cleanup	229
Dynamic Thread Cleanup	123
Restore Next Thread	85
Context Switch	184

Table 1. Number of cycles necessary to perform thread-related operations.

Since children threads may need access to global variables associated with a loadable binary, TinyLD terminates the binary only when all of its children have also terminated. To ensure this, we designed a custom synchronization primitive, we call a *blocking reference counter*. As described above, every thread spawned by the root thread or one of its children inherits a pointer to the original `init_block`. This block contains a reference counter that increments when a new thread is spawned and decrements whenever a thread terminates. When the root thread itself finishes, it blocks until this reference counter reaches zero. At that point, the root thread de-allocates any resources associated with the program and marks the flash ROM segments in which the program was stored as free.

5 Evaluation

Our primary goals in designing TOSThreads were to architect a system that was both thread-safe and non-invasive with respect to the timing of critical OS services running in the kernel. Additionally, we wanted a system that was backwards compatible with TinyOS, supported an evolvable kernel, and enabled dynamic linking and loading of applications at runtime. In this section, we evaluate how well TOSThreads meets these requirements.

We first measure microbenchmarks based on the cycle counts of TOSThreads basic scheduler operations. Second, we analyze a representative sensor network application as well as one that has a long-running compression computation, examining how TOSThreads can simplify programs and quantifying its energy cost. Third, we evaluate dynamic linking and loading and the MicroEXE file format using code size as a metric, both in terms of bytes and lines of code. Finally, we evaluate TOSThreads' expressiveness by presenting a reimplementation of the Tenet API using TOSThreads as well as Latte, a novel JavaScript dialect.

All measurements use the Tmote Sky platform running at 4MHz with a serial baud rate of 57,600 bps. We use the onboard temperature, humidity, total solar, and photo active radiation sensors for experiments including sensor readings.

5.1 Microbenchmarks

Tables 1 and 2 present the number of cycles necessary to perform all relevant thread scheduler and basic synchronization operations, respectively. With the Tmote Sky running at 4 MHz, these operations (with the exception of starting a dynamic thread) take less than a few hundred cycles to complete. These numbers translate to less than 70 μsec of computation time per operation. Even starting a dynamic thread (which can take as many as 800 cycles, depending on the duration of `malloc()`), takes less than 200 μsec. Thus, the cost of performing these operations is negligible in terms of their impact on the system's responsiveness.

Operation	Number of cycles
Mutex Init	13
Mutex Lock	17
Mutex Unlock	71
Barrier Reset	13
Barrier Block	41
Barrier Block with Wakeup	$6 + 302 \times num_waiting$
Condvar Init	8
Condvar Wait	30
Condvar Signal Next	252
Condvar Signal All	$314 \times num_waiting$
Refcount Init	12
Refcount Wait On Value	39
Refcount Increment	11
Refcount Decrement	11
Refcount Inc/Dec with Wakeup	$11 + 320 \times num_waiting$
Join Block	74
Join Wakeup	$74 + 326 \times num_waiting$

Table 2. Number of cycles necessary to perform the basic TOSThreads synchronization primitives.

Tables 1 and 2 do not represent the true application-level cost of using TOSThreads, as more than one of these operations are usually performed in sequence. For example, whenever a thread is suspended, either via a blocking system call or because it waits on a synchronization primitive, it must be explicitly woken up before it resumes. The *total* cost of suspending the thread must then be calculated as the sum of the suspend, context switch, and wakeup costs, for a total of 344 cycles.

The total suspend cost is relevant when calculating the *total* overhead of making a blocking system call. The first column of Table 3 shows the marginal overhead of making a blocking system call, while the second column presents the total overhead including the cost of suspending and resuming a thread (i.e., adding 344 cycles). In turn, these totals are relevant when measuring the energy cost of using TOSThreads, which we present next.

5.2 Energy Analysis

To measure the impact of TOSThreads on energy consumption, we implement a representative sensor network application and calculate the energy overhead of performing all system calls, context switches, and thread synchronization operations. Specifically, we develop a 'Sense, Store, and Forward' (SSF) application consisting of producer threads which sample sensors once every logging period and log their values to flash memory. The application also includes a consumer thread which reads the values written to flash and transmits them over the radio, using a different sending period. Our SSF application has six threads: one for sampling each of the four sensors onboard the Tmote Sky, one for logging these sensor values to flash, and one for sending them over the radio. We set the logging period to 5 minutes and the sending period to 12 hours, resulting

Operation	Number of cycles		Operation	Number of cycles	
	Marginal	Total		Marginal	Total
Sleep	371	715	Log Sync	466	810
			Log Seek	476	820
StdControl Start	466	810	Log Read	491	835
StdControl Stop	466	810	Log Append	500	844
			Log Erase	468	812
AM Send	390	734	Block Sync	468	812
AM Receive	912	1256	Block Read	495	839
			Block Write	495	839
Sensirion					
Sensor Read	277	621	Block Erase	466	810
ADC Read	477	821	Block CRC	506	850

Table 3. Overhead of invoking the system calls that the standard TOSThreads C API provides. The marginal column includes the cost of the system call itself while the total column includes the cost of invoking the underlying thread scheduler operations.

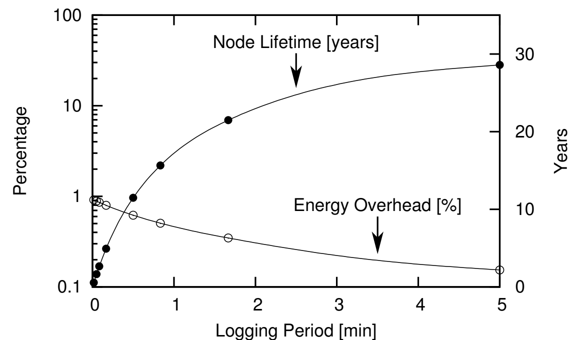

Figure 5. Energy overhead of implementing SSF application using TOSThreads as a function of the logging period. Also shown is the node lifetime, using standard AA batteries as the energy source.

Program	Number of symbols	Number of patched addresses
Null	0	0
Blink	11	15
RadioStress	37	43
SenseSF	50	93
BaseStation	47	70

Table 4. The number of symbols and the number of addresses that TinyLD patches for five sample applications.

in 144 samples gathered during each sending period. The six threads synchronize using a combination of mutexes and barriers.

To calculate the energy overhead of executing this application, we combine the system call costs found in the second column of Table 3 and the synchronization costs calculated in Table 2. Specifically, for each logging period we include the cost of two Sensirion Sensor Reads, two ADC Reads, four Mutex Locks (plus 344 cycles for suspends), four Mutex Unlocks, eight Barrier Blocks, one Log Write, and one Sleep call for a total of 6,286 cycles (log_cost). The overhead during each sending period is the sum of 144 Log Reads, 144 AM Sends, 144 Mutex Locks (plus suspends), 144 Mutex Unlocks, and one Sleep operation, for a total of 288,859 cycles (send_cost).

As measured in [21], the current the MSP430 processor draws while active is 1.92 mA. Using this value, the total energy consumed during each log and send period is:

$$\begin{aligned} E_{\text{log_cost}} &= (\text{log_cost} \times 1.92mA)/4MHz \\ &= 2.87\mu As \\ E_{\text{send_cost}} &= (\text{send_cost} \times 1.92mA)/4MHz \\ &= 132.23\mu As \end{aligned}$$

Using an analysis similar to the one in [21], we calculate the total lifetime of this application with different logging periods[2]. In all cases we adjust the sending period to be 144 times the logging period (derived from a typical 12 hour sending period and 5 minutes sampling interval). Figure 5 presents the percentage of energy consumed by system calls and thread synchronization primitives as a function of the logging period. In all cases this cost is less than 1%.

5.3 Supporting Long-Running Computations

We evaluate the ability of TOSThreads applications to perform long-running computations without interfering with the responsiveness of the underlying TinyOS kernel. To do so, we compare two versions of an application that uses compression: one implemented using standard TinyOS tasks and another using TOSThreads. In both cases, the application receives packets over the serial port every 50 msec and buffers their payloads in RAM (25 bytes per packet). Whenever the buffer is full, the application compresses the entire content of the buffer (1,250 bytes) with the Lempel-Ziv-Welch (LZW) compression algorithm. Experimental results show that compressing the buffer requires approximately 1.4 sec, which is more than sufficient to represent a long-running computation; any operation that lasts longer than 50 msec results in an unresponsive system that will start dropping packets.

The metric we use for this experiment is the total number of packets dropped after 500 serial packets have been sent. Since TinyOS does not support task preemption, we expect that the TinyOS version of the program will drop multiple packets while compressing its buffer. The experiment confirmed our expectation: TinyOS dropped 127 packets while TOSThreads dropped zero.

Although this application does not necessarily reflect the actual long-running computations we expect motes to perform, the results we provide expose a fundamental limitation in the existing TinyOS concurrency model – running long computations severely affects its performance. TOSThreads removes this limitation.

5.4 Dynamic Linking and Loading

TinyLD introduces space overhead in terms of application and system code size, as well as execution overhead in terms of the time necessary to link and load an application binary. In this section, we evaluate these overheads by measuring the cost of dynamically loading five sample applications compiled into the MicroExe format: `Null`, `Blink`, `Radio Stress`, `SSF`, and `BaseStation`. The first appli-

[2] This analysis assumes that the mote is powered by two AA batteries with an approximate capacity of 2,700 mAh ($9.72 \cdot 10^9 \mu$As).

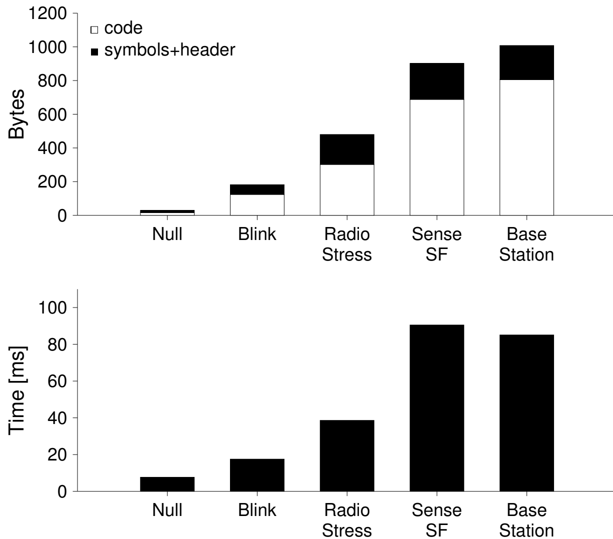

Figure 6. TinyLD overhead for five sample applications. The top graph shows the size of each application using the MicroExe format and the bottom graph presents the time TinyLD requires to link and load each application.

cation is effectively empty and serves as a baseline for the fixed cost of linking and loading. `Blink` is the standard TinyOS application that repeatedly blinks a mote's LEDs, while `RadioStress` transmits radio packets as fast as possible. Finally, `SSF` is the application described in Section 5.2, and `BaseStation` is the standard TinyOS application that forwards radio packets to the serial port (and vice-versa).

The size of a MicroExe binary depends on four factors: the size of the machine code ($Code$), the total number of relocations (U_{Reloc}), the total number of allocations (U_{Alloc}), and the total number of initialized global variables (U_{Init}). Since MicroExe stores patched addresses as chained references, U_{Reloc} is equal to the number of unique symbols in the program (please refer to [30] for more details). The size of a MicroExe file is then given by:

$$Code + (U_{Reloc} + U_{Alloc}) \cdot 4 + U_{Init} \cdot 6 + 5 \cdot 2$$

The graph at the top of Figure 6 shows the breakdown of the MicroExe binary into its code and header components for each of the five sample applications.

The time required to link and load a MicroExe binary depends on multiple factors. First, TinyLD must copy the entire machine code section of a binary to the MCU's flash ROM. Experiments show that copying two bytes of data from memory to flash ROM takes 188 cycles (47 μsec on the Tmote Sky). Second, the loading time depends on both the number of unique symbols in the binary and the number of addresses that TinyLD must patch. This is because TinyLD implements an iterative loading process whereby the number of unique symbols determines the time required to find the next smallest patched address, and the number of patched addresses determines the total number of iterations required.

Table 4 presents the total number of symbols and addresses that require patching in each of the five sample ap-

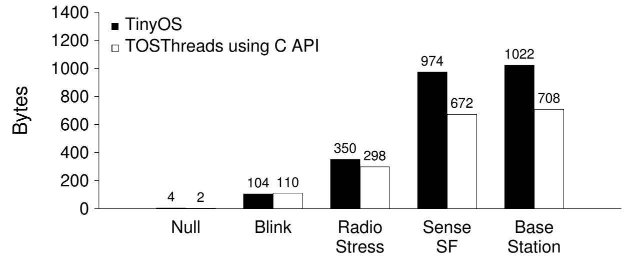

Figure 7. Comparison of application code sizes for five sample applications implemented using standard TinyOS and TOSThreads.

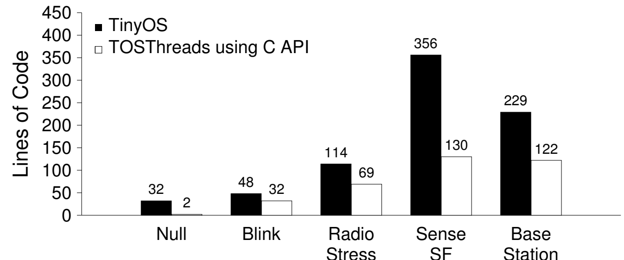

Figure 8. Comparison of application lines of code for five sample applications implemented using standard TinyOS and TOSThreads.

plications. The graph at the bottom of Figure 6 presents the linking and loading time for these applications. One observation is that although SSF is smaller than BaseStation in terms of binary size, it takes longer to load because it has more symbols and patched addresses (see also Table 4).

5.5 Code Size

We also compare the code size of just the *application* portion of our sample applications when implemented in both standard TinyOS and TOSThreads. As Figures 7 and 8 indicate, the TOSThreads versions are more compact in terms of both binary code size and lines of code. We gathered binary code sizes by manually counting the application-specific portion of the binary resulting from an `msp430-objdump`. We counted lines of code using a version of SLOCCount [38], modified to recognize nesC.

Finally, we present a breakdown of the binary code size and RAM usage of a complete TOSThreads kernel compiled together with TinyLD (Figure 9). This kernel implements the standard C API (i.e, one providing standard services such as sending packets, sampling sensors, and writing to flash).

5.6 Tenet

We have re-implemented the Tenet API using TOSThreads. Tenet applications specify tasks as linear dataflow programs consisting of a sequence of *tasklets*[3]. Each tasklet is implemented as a core TinyOS component, providing a specific TinyOS service. For example, an application that

[3]Even though Tenet runs on top of TinyOS, *Tenet tasks* are logically distinct from *TinyOS tasks*.

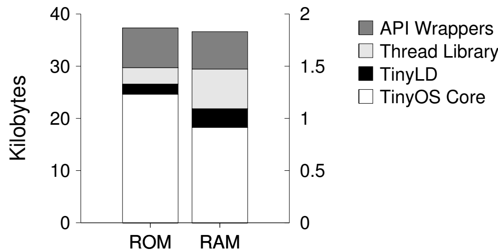

Figure 9. Breakdown of the binary code size and RAM usage of a complete TOSThreads kernel based on the standard C API compiled together with TinyLD.

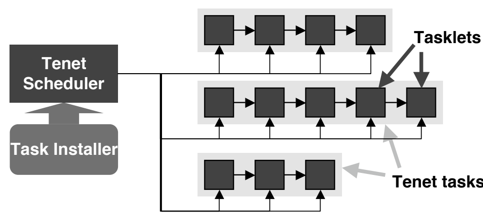

Figure 10. Original Tenet: Tenet scheduler executes Tenet tasks by scheduling the execution of each tasklet included in those Tenet tasks.

wants to be notified when the temperature at any mote exceeds 50°F would write the following task:

```
Repeat(1000ms) -> Sample(ADC1,T)
    -> LEQ(A,T,50) -> DeleteDataIf(A) -> Send()
```

Tenet consists of a *task library*, a *task installer*, and a *task scheduler*. The task library contains a collection of tasklets, the task installer dynamically executes tasks it receives from the network, and the task scheduler coordinates the execution of all running tasks. This scheduler maintains a queue of pending tasks and services them in round-robin order (see Figure 10). Each tasklet runs to completion before the next one is scheduled. Furthermore, Tenet includes a task dissemination protocol, transport and routing protocols, a time-synchronization protocol [25], and several other custom TinyOS components for accessing sensors and timers.

Tenet-T is a reimplementation of Tenet that uses the TOSThreads library. Specifically, Tenet-T spawns one thread to service each Tenet task and replaces Tenet's original task scheduler with the TOSThreads thread scheduler. However, Tenet-T maintains the same set of tasklet APIs defined in the original Tenet and also uses the same task installer as in the original Tenet. The rest of the original Tenet code runs unmodified but now becomes part of the *kernel*, running inside the TinyOS thread. Figure 11 provides a pictorial overview of these modifications.

The main difference between Tenet-T and Tenet is that TOSThreads supports preemption. This allows each thread running a Tenet task to execute its tasklets one after another without regard for how long each of them might take. In the original Tenet, the Tenet task scheduler has to interleave tasklets from multiple tasks in order to keep them all responsive; individual tasklets run to completion and cannot perform long running computations. *Tenet-T* removes this limitation at the expense of a small increase in code size (see Figure 13).

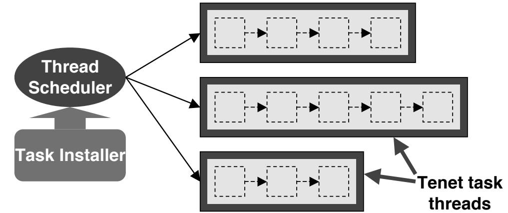

Figure 11. Tenet-T: Each Tenet task is a TOSThreads user thread and the thread scheduler schedules the execution of each task.

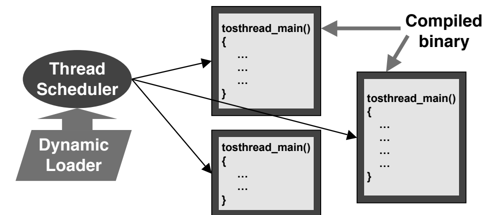

Figure 12. Tenet-C: Each Tenet task is a native mote binary, compiled from C code, that runs as a user thread. Binaries are loaded by TinyLD and scheduled by the TOSThreads scheduler.

Tenet-C is a reimplementation of Tenet that *significantly increases the expressivity of the tasking language*, yet does not require drastic modifications to the overall system. In *Tenet-C*, the user writes a C program, instead of a data-flow task description, and compiles it into a dynamically loadable binary object. For example, the Tenet temperature sensing task example shown before, can be re-written as:

```
void tosthread_main(void* arg) {
    uint16_t T;
    for(;;) {
        tosthread_sleep(1000ms);
        T = Sample(ADC1);
        if (T <= 50)
            continue;
        Send(&T, sizeof(T));
    }
}
```

More complex applications can also be written quite easily in Tenet-C.

The *Tenet-C* kernel is identical to that of *Tenet-T* except that it uses TinyLD to dynamically link and load application binaries (Figure 12). However, *Tenet-C*'s API is significantly smaller. In *Tenet-C*, we only need to implement tasklets such as Sample, Get, and Send in the form of blocking system calls into the Tenet kernel. Many of the other Tenet tasklets provide functionality (e.g., arithmetic operations, comparisons) which is already provided natively by C. In fact, the C language constructs for some of these functions are strict supersets of those Tenet's tasking language provides. For example, the original Tenet had no support for branching, and limited support for looping. An additional

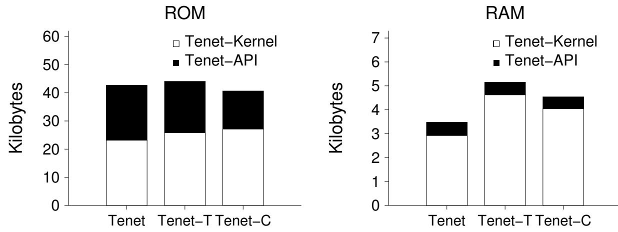

Figure 13. Tenet mote code size (ROM/RAM). Tenet-C is a reimplementation of Tenet which dynamically loads and executes compiled binary tasks as user threads.

benefit of Tenet-C is that the binary code size is smaller than that of the original Tenet, as Figure 13 suggests.

All three implementations (Tenet, Tenet-T, Tenet-C) have been tested on a 35-node testbed using five simple Tenet applications: `blink`, `pingtree` (gathers topology information and draws routing tree), `system` (gathers mote's internal system information), `collect` (periodically collects sensor data), and `deliverytest` (tests end-to-end reliable packet delivery). All application binaries were disseminated to motes in the network using Tenet's internal task dissemination protocol. These implementations can all run multiple applications concurrently and their kernels contain comparable functionality.

Finally, we report on an experiment that illustrates the benefits of preemptive threading for task execution in Tenet-C. In this experiment, we ran two Tenet tasks concurrently:

```
TaskLong  : Repeat(100ms) -> ReadBlock(A,1024)
              -> Avg(B,A) -> MeanDev(C,A) -> Send()

TaskSample: Repeat(50ms) -> TimeStamp(A)
              -> Sample(ADC5,V) -> Send()
```

`TaskLong` periodically repeats a long running computation (taking the mean deviation of 1024 samples) that takes longer than 10ms, while the `TaskSample` task samples a sensor every 50ms.

Figure 14 plots the distribution of the measured inter-sample intervals for `TaskSample`, for both Tenet and Tenet-C, both when that task runs by itself and when it runs concurrently with `TaskLong`. The expected inter-sample interval in `TaskSample` is 50ms. In the original Tenet, which contains a cooperative scheduler, the long-running computation cannot be preempted. This induces significant sample timing jitter for `TaskSample`; sometimes two consecutive samples are missed! However, the jitter in Tenet-C is much less, with a small number of samples experiencing a jitter of up to 8 ms. Some of the jitter is introduced by thread scheduling overhead, since `TaskSample` experiences a 1ms jitter for about a fifth of the samples even when run alone. This experiment validates the design of preemptive schedule in TOSThreads, and illustrates how it can correctly support timing-sensitive applications in the presence of long-running computations.

5.7 Additional uses of TOSThreads

TOSThreads can be used to simplify the implementation of high-level programming languages for motes. Latte [31]

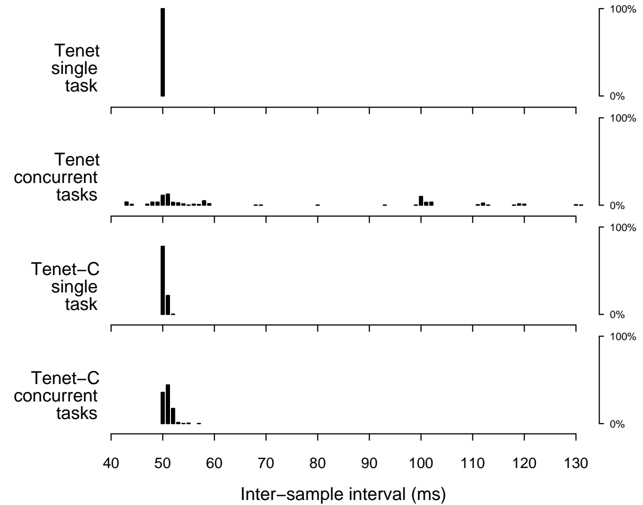

Figure 14. Distribution of measured inter-sample interval of a simple periodic sampling task, for Tenet and Tenet-C, when running by itself or running concurrently with another task which has long-running computation. Intended inter-sample interval is 50 milliseconds.

is a Javascript variant designed to simplify the writing of efficient sensor network applications. Latte programs can either be interpreted within a JavaScript-enabled web browser or compiled directly down to C. Running programs in a browser simplifies the early stages of application development and helps to reduce debugging cycles.

In our TOSThreads-based implementation of Latte, programs compiled into C make system calls that are either statically-linked against a TinyOS kernel or dynamically loaded onto a running mote using TinyLD. In the absence of the blocking API that TOSThreads provides, a Latte implementation on TinyOS would have had to expose an event-driven programming interface, increasing the compiler complexity and reducing ease-of-use. Indeed, the original implementation of Latte on TinyOS had an event-driven interface for exactly this reason.

TOSThreads has also been successfully used to ease the implementation of a polling-based SD card and GPS driver for the MAMMARK [11] project at UCSC, as well as for upcoming versions of the SPINE body sensor network project from Telecom Italia [19].

6 Related Work

We review prior threading proposals for TinyOS and other sensor network operating systems, as well as prior work on dynamic linking and loading techniques used on mote class devices. TOSThreads builds on this prior work, enabling preemptive threads through message passing on top of an event-based kernel.

Many thread-based sensor network operating systems rely on kernel traps to achieve synchronization at the cost of limited concurrency. For example, TinyMOS [35] runs TinyOS inside a dedicated thread just as TOSThreads does. However, it requires TinyOS code to be instrumented with syn-

chronization primitives around all core OS abstractions, as the TinyOS concurrency model does not understand preemption outside of interrupts. TOSThreads, on the other hand, is able to avoid these problems through its use of message passing. This seemingly small change to the model allows arbitrary concurrency in the kernel, and requires only minimal changes to the existing TinyOS code base (in the interrupt handler post-ambles and the boot sequence).

Cooperative threading is another popular approach used by many threads packages such as TinyThreads [27]. However, as discussed in Section 2, cooperative threading requires each thread to voluntarily yield the processor, placing the burden of when to do so on the application programmer. In the case of TinyThreads, a single long-running thread could disrupt the TinyOS task queue and therefore affect the timing of critical kernel services. In contrast, TOSThreads requires no explicit yields, simplifying programming and preventing errors: users can run multiple infinite loops.

Unlike Protothreads, which do not maintain thread context across blocking calls [10], TOSThreads is a full threads implementation. Therefore, users do not have to manually maintain continuations in the form of global variables, simplifying program design. On the other hand, this also means that TOSThreads requires more memory than Protothreads, to maintain stacks.

Numerous other concurrency proposals for TinyOS exist, including fibers [36], virtual machine threads [24], and preemptive tasks [7]. While TOSThreads borrows some of its ideas from many of these approaches, none of these existing techniques allow users to write simple, thread-based programs on a TinyOS kernel; they either limit the number of threads available on the system (fibers), are built into a specialized runtimes (fibers, VM threads), or break the TinyOS concurrency model (preemptive tasks).

In addition to fully event-driven and Protothread programming models, Contiki provides an optional cooperative threading library to applications. Although initial work introduced preemptiveness [10], synchronization mechanisms for making the Contiki kernel thread-safe were never added to the Contiki code.

RETOS [4] is a sensor network operating system designed from the ground up to support the POSIX-like threading API. While RETOS's implementation of the POSIX standard is not 100% compatible with standard POSIX, it provides enough functionality to ease a programmer's learning curve. Like TOSThreads, RETOS also separates the kernel from the user applications using a set of system calls. However, in contrast to TOSThreads, RETOS does not have support for events and relies completely on blocking calls.

Nano-RK is a real-time operating system designed for wireless sensors networks [12]. The OS is based on the notion of reservations and it implements preemptive multitasking with fixed priorities capable of guaranteeing that task deadlines are met. Nano-RK uses message passing to implement signals, but its support for events (i.e., timers and external interrupts) is limited.

LiteOS [3], which provides a UNIX-like shell interface, is also based on a message-passing architecture. LiteOS application threads trigger a kernel thread to run and handle system calls. However, unlike TOSThreads, many of the system calls in LiteOS block on spin loops, constraining its overall concurrency.

Other embedded OSs such as uC/OS [28] and FreeRTOS [1], also use message passing, but differ from TOSThreads in one important respect. TOSThreads uses message passing as the interface for making system calls between application threads and the kernel, while these other systems use them for communication between threads at the application level.

Message passing systems like this are not new: they are a 30-year old abstraction [22] and a staple of microkernel designs such as Mach [32]. Traditional microkernels typically have kernel threads independent of user threads, which respond to application requests. Separating kernel and user concurrency in this way enables the kernel to control reentrancy without explicit synchronization: instead, synchronization occurs around the message queues between user and kernel threads. While this approach has architectural elegance, experience has shown it to be prohibitively expensive: early implementations (e.g., MkLinux) exhibit up to a 60-fold slowdown on some system calls and even state-of-the art microkernels such as L^4Linux exhibit slowdowns of 20-150% [16]. Virtual memory is a major cause of this slowdown as well as the high cost of system calls in multithreaded operating systems generally. TOSThreads' contribution in this area is to apply the use of message passing to TinyOS in a way that allows fully preemptive threads while requiring only minor changes to existing TinyOS code.

In terms of prior work on the dynamic linking and loading techniques used in TOSThreads, there have been other proposals for dynamically loading binaries on mote platforms. FlexCup also allows dynamic loading of TinyOS components [26]. However, FlexCup uses a linking and loading method that requires rebooting the node for the new image to run. Moreover, the application halts during the linking and loading process. TinyLD does not have these limitations.

With Contiki, Dunkels et al. [8] introduced the Compact ELF (CELF) binary format to quantify the overhead of dynamic loading and its relation to the standard ELF format. Compact ELF (CELF) binaries, like MicroExe binaries, are also compressed versions of ELF, though neither format is backwards compatible with it. The only real difference between MicroExe and CELF is its use of chained references which makes the binary image linear in the number of symbols rather than the number of relocations.

7 Summary and Conclusions

TOSThreads is a fully functional thread library designed for TinyOS. It provides a natural extension to the existing TinyOS concurrency model, allowing long running computations to be interleaved with timing-sensitive operations. TOSThreads' support for efficiently running dynamically loaded binaries, combined with its ability to support a flexible user/kernel boundary, enables experimentation with a variety of high-level programming paradigms for sensor network applications. We hope that this capability will accelerate the movement towards a standard TinyOS kernel that can support a wide range of applications.

Modern threading systems and kernels are optimized for high-performance processors, with virtual memory, caches, and large context switch latencies. In contrast, TOSThreads is designed for a microcontroller, whose different properties cause an approach discarded long ago – message passing – to be both efficient and compelling. This suggests another way in which different application workloads and hardware considerations cause system design in ultra-low power sensor networks to differ from that in mainstream platforms.

Acknowledgments

We would like to thank our shepherd, Adam Dunkels, and the anonymous reviewers for their help improving our paper. This material is based upon work partially supported by the National Science Foundation under grant #0754782 ("IDBR"), #0121778 ("CENS"), and #0520235 ("Tenet").

8 References

[1] R. Barry. FreeRTOS, a FREE open source RTOS for small embedded real time systems. Available at http://www.freertos.org.

[2] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. *ACM/Kluwer MONET, Special Issue on Wireless Sensor Networks*, 10(4):563–579, Aug. 2005.

[3] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He. The LiteOS Operating System: Towards Unix-Like Abstractions for Wireless Sensor Networks. In *Proc. of IPSN'08*, 2008.

[4] H. Cha, S. Choi, I. Jung, H. Kim, H. Shin, J. Yoo, and C. Yoon. RETOS: resilient, expandable, and threaded operating system for wireless sensor networks. In *Proc. of IPSN'07*, 2007.

[5] K. Chintalapudi, T. Fu, J. Paek, N. Kothari, S. Rangwala, J. Caffrey, R. Govindan, E. Johnson, and S. Masri. Monitoring Civil Structures with a Wireless Sensor Network. *IEEE Internet Computing*, 10(2), March/April 2006.

[6] Crossbow Inc. Imote2: High-Performance Wireless Sensor Network Node. Available at: http://www.xbow.com/Products/Product_pdf_files/Wireless_pdf/Imote2_Datasheet.pdf, 2007.

[7] C. Duffy, U. Roedig, J. Herbert, and C. J. Sreenan. Adding Preemption to TinyOS. In *Proc. of EmNets*, 2007.

[8] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proc. of SenSys'06*, 2006.

[9] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proc. of Emnets-I*, 2004.

[10] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proc. of SenSys'06*, 2006.

[11] G. Elkaim, E. Decker, G. Oliver, and B. Wright. Marine mammal marker (mammark) dead reckoning sensor for in-situ environmental monitoring. In *Proc. of the ION/IEEE PLANS*, 2006.

[12] A. Eswaran, A. Rowe, and R. Rajkumar. Nano-RK: An Energy-Aware Resource-Centric RTOS for Sensor Networks. In *Proc. of RTSS'05*, 2005.

[13] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proc. of PLDI*, 2003.

[14] O. Gnawali, B. Greenstein, K.-Y. Jang, A. Joki, J. Paek, M. Vieira, D. Estrin, R. Govindan, and E. Kohler. The TENET Architecture for Tiered Sensor Networks. In *Proc. of SenSys'06*, 2006.

[15] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A Dynamic Operating System for Sensor Nodes. In *Proc. of Mobisys'05*, 2005.

[16] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ-kernel-based systems. In *Proc. of SOSP*, 1997.

[17] J. Hicks, J. Paek, S. Coe, R. Govindan, and D. Estrin. An Easily Deployable Wireless Imaging System. In *Proc of ImageSense'08*, 2008.

[18] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. In *Proc. of ASPLOS'00*, 2000.

[19] T. Italia. Spine: Signal processing in node environment. Available at http://spine.tilab.com/.

[20] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon. Health Monitoring of Civil Infrastructures Using Wireless Sensor Networks. In *Proc. of IPSN'07*, 2007.

[21] K. Klues, V. Handziski, C. Lu, A. Wolisz, D. Culler, D. Gay, and P. Levis. Integrating Concurrency Control and Energy Management in Device Drivers. In *Proc. of SOSP*, 2007.

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

[23] J. R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers, 2000.

[24] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *Proc. of NSDI*, 2005.

[25] M. Marot, B. Kusy, G. Simon, and A. Ledeczi. The flooding time synchronization protocol. In *Proc. of SenSys'04*, 2004.

[26] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. FlexCup: A Flexible and Efficient Code Update Mechanism for Sensor Networks. In *Proc. of EWSN'06*, 2006.

[27] W. P. McCartney and N. Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In *Proc. of SenSys'06*, 2006.

[28] Micrium. uc/os-ii, the real-time kernel. Available at http://www.micrium.com/page/products/rtos/os-ii/.

[29] MoteIV Corporation. Tmote Sky. Available at: http://www.moteiv.com/products/tmotesky.php.

[30] R. Musăloiu-E., C.-J. M. Liang, and A. Terzis. A Modular Approach for Developing and Updating Wireless Sensor Network Applications. Technical Report 21-10-2008-HiNRG, Johns Hopkins University, 2008.

[31] R. Musăloiu-E. and A. Terzis. The Latte Language. Technical Report 22-10-2008-HiNRG, Johns Hopkins University, 2008.

[32] R. Rashid, D. Julin, D. Orr, R. Sanzi, R. Baron, A. Forin, D. Golub, and M. Jones. Mach: a system software kernel. In *Proc. of COMPCON'89*, 1989.

[33] C. Sadler and M. Martonosi. Data compression algorithms for energy-constrained devices in delay tolerant networks. In *Proc. of SenSys'06*, 2006.

[34] The TinyOS Community. The tinyos documentation wiki. Available at: http://docs.tinyos.net.

[35] E. Trumpler and R. Han. A Systematic Framework for Evolving TinyOS. In *Proc. of EmNetS-III*, 2006.

[36] M. Welsh and G. Mainland. Programming Sensor Networks Using Abstract Regions. In *Proc. of NSDI*, 2004.

[37] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and Yield in a Volcano Monitoring Sensor Network. In *Proc. of OSDI*, 2006.

[38] D. Wheeler. The SLOCCount utility. Available at http://www.dwheeler.com/sloccount/.