TOSThreads: Extending the TinyOS Concurrency Model to Support Preemption

Kevin Klues⋆, Chieh-Jan Mike Liang†, Jeongyeup Paek‡, Razvan Musaloiu-E†, Ramesh Govindan†, Philip Levis⋆, Andreas Terzis†

* Stanford University † Johns Hopkins University ‡ University of Southern California
Stanford, CA Baltimore, MD Los Angeles, CA

ABSTRACT

We present TOSThreads, an application-level threads package for TinyOS. In contrast to previous threads packages proposed for TinyOS, TOSThreads supports fully preemptive threads while maintaining the existing TinyOS concurrency model. TOSThreads defines a conceptual user/kernel boundary, whereby TinyOS runs as a high-priority kernel thread and application threads execute at lower priority. This approach naturally supports long-running computations while preserving the timing-sensitive nature of TinyOS itself. Additionally, the existence of a user/kernel boundary enables dynamic linking and loading of application binaries at runtime. The API at this boundary defines the set of TinyOS services supported by a TOSThreads kernel and is customizable in support of a diverse set of applications.

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 90ms for a representative sensing application. We compare different programming models built on top of TOSThreads, including standard C with blocking system calls and a reimplementation of Tenet. Additionally, we evaluate the ability of TOSThreads to run computationally intensive tasks. Taken as a whole, these results suggest that TOSThreads is an important step towards defining a TinyOS kernel that can support long-running computations and high concurrency.

1. INTRODUCTION

Many mote operating systems use event-driven execution to support multiple concurrent execution contexts with the memory cost of a single stack [6, 11, 13]. 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.

On the other hand, preemptive threading offers a more intuitive programming paradigm for developing higher-level services and applications. In this model, application writers need not explicitly manage yield points or continuations, or partition a long running computation to avoid missing events. Compression is a concrete example of an operation that can benefit from threading. Many sensor network applications, such as seismic sensing [27] and structural monitoring [2, 15] could benefit greatly from data compression. Nevertheless, real sensor network deployments rarely use it due to the difficulty of implementing it in event-driven environments.

In this paper, we explore this tension between being able to manage concurrency in the face of resource constraints and having an intuitive programming model. Specifically, we describe a new threads package called TOSThreads for TinyOS that has several novel goals:

**Fully preemptive threads:** Application programmers should not have to manually manage yield points or continuations, greatly simplifying application development.

**Minimal disruption:** Adding threads should not negatively affect the OS’s performance and changes required to the existing code should be highly localized and minimal. This goal enables system developers to achieve high performance and concurrency wherever necessary.

**Flexible boundary:** Developers must be able to experiment with different “kernels” by altering the boundaries between threaded and event-driven code, based on tradeoffs between ease of use, efficiency, and application requirements.

**Flexible application development:** The system must enable programmers to develop their applications using multiple programming paradigms and provide a way to dynamically link and load executable application code.

To the best of our knowledge, no existing thread package for TinyOS satisfies all of these requirements. TinyThreads [21] relies on applications to explicitly yield the processor. TinyMOS [25] runs TinyOS inside a dedicated Mantis [1] thread, but requires placing a lock
around most of the TinyOS code, limiting overall concurrency and efficiency. While our work has been inspired by these approaches, TOSThreads is different as it provides true multi-threading in TinyOS without limiting performance or sacrificing its event-driven programming model.

Contributions. We identify three contributions related to the goals we have just set forth. (1). We provide TOSThreads, a fully-preemptive threads library for TinyOS. In TOSThreads, TinyOS runs inside a high priority kernel thread, while all application logic resides inside user-level threads, which execute whenever TinyOS becomes idle. This approach naturally extends the existing TinyOS concurrency model while adding support for long-running computations. (2). TOSThreads applications access underlying TinyOS services through a kernel API of blocking system calls. This framework allows system developers to evolve the kernel API by wrapping blocking system calls around event-driven TinyOS services. Moreover, by exporting this API in both nesC [9] and ANSI C, we allow developers to implement efficient applications without the need to learn a new language. (3). TOSThreads enables on-demand execution of application threads received from the network as binary objects. TinyLD, our dynamic linker/loader, patches a binary’s unresolved system call references and loads it into the mote’s memory before executing it.

In many ways, the TOSThreads approach resembles that taken in Contiki, where the Contiki core runs in one thread and additional threads context switch with it. Our contribution beyond this work is to propose message passing between the application threads and the core OS thread, thereby avoiding the race conditions typical to preemptive threads that directly call kernel code. The challenges such race conditions present perhaps explain why no Contiki platform currently supports thread preemption [4].

Together, user-level threads, dynamic application loading and linking, and a flexible blocking system call API, greatly simplify existing programming abstractions for TinyOS and the implementation of new ones. While these facilities are, of course, available on general-purpose systems today, they have not previously been demonstrated together in the context of the TinyOS event-driven operating system. To substantiate TOSThreads’s flexibility we have reimplemented the Tenet programming system [10], on top of TOSThreads. The resulting system has higher expressivity without increasing code size. TOSThreads has also enabled the development of a novel programming language, Latte, a JavaScript dialect that compiles to C. Finally, our evaluation results show that TOSThreads generates minimal overhead in terms of execution speed and energy consumption, while efficiently supporting long-running computations.

2. ARCHITECTURE

The existing TinyOS concurrency model has two execution contexts: synchronous (tasks) and asynchronous (interrupts). These two contexts follow a strict priority scheme: asynchronous code can preempt synchronous code but not vice-versa. TOSThreads extends this concurrency model to provide a third execution context in the form of user-level application threads. While application threads cannot preempt either synchronous or asynchronous code, they can preempt each other. Application threads synchronize using standard synchronization primitives such as mutexes, semaphores, barriers, condition variables, and blocking reference counters (a mechanism we have developed, §3.4).

Figure 1 presents the TOSThreads architecture, consisting of five key elements: the TinyOS task scheduler, a single kernel-level TinyOS thread, a thread scheduler, a set of user-level application threads, and a set of system call APIs and their corresponding implementations. Any number of application threads can concurrently exist (barring memory constraints), while a single kernel thread runs the TinyOS task scheduler. The thread scheduler manages the concurrency between application threads, while a set of system calls provides access to the TinyOS kernel.

In order to preserve the timing-sensitive nature of TinyOS, the kernel thread has higher priority than application threads. So long as the TinyOS task queue is non-empty, this TinyOS thread takes precedence over all application threads. Once the TinyOS task queue empties, control passes to the thread scheduler and application threads can run. The processor goes to sleep only when either all application threads have run to completion, or when all threads are waiting on synchronization primitives or blocked on I/O operations.

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 have higher priority than the TinyOS thread, the TinyOS thread will not wake up to process the task until after the interrupt handler has completed; this sequence of operations is identical to that of traditional TinyOS without TOSThreads support. Because interrupts can arrive at anytime, however, the TinyOS thread may require a context switch with an interrupted application thread. Control eventually returns to the application thread after the TinyOS thread has emptied the task queue.

**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. Our evaluations show these modifications introduce minimal disruption to the operation of TinyOS (§4).

**Flexible User/Kernel Boundary:** A primary difference between TOSThreads and other TinyOS threading implementations is that TOSThreads defines a flexible boundary between user 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. The API that a kernel ultimately provides depends on the set of TinyOS services its designer wishes to present to applications.

Through its 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 (§3.3) and one that implements the Tenet API (§4.6).

**Dynamic Linking and Loading:** Defining a 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 some predefined method (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.

### 3. 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 for performing a context switch and (2) adding a post-amble to every interrupt handler. Defining these functions is a fairly straightforward process, and implementations exist for many popular TinyOS platforms. As of TinyOS 2.1.0, TOSThreads is part of the baseline TinyOS distribution, with support for Tmote Sky, Mica2, Mica2dot, MicaZ, Iris, eyesIFX, Shimmer, and TinyNode motes. Full source code can be found at [http://www.tinyos.net/](http://www.tinyos.net/).

#### 3.1 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 commands.
above, executes in the context of the calling thread; only
the interrupt handler post-amables 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 la-
tency across multiple application-level computing tasks.
While application threads currently run with the same
priority, one can easily modify the scheduler to support
other 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 trig-
ger the normal TinyOS boot sequence. Once TinyOS
boots and processes all of its initial tasks, control re-
turns 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 differ-
ent queues. Calling interrupt() or suspend() places a
thread onto one of these queues while calling wakeup() removes it from a queue.

3.2 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 pro-
vided by preemptive threading outweigh these costs in
many situations. That said, one can use techniques such
as those proposed by McCartney and Sridhar [21] to es-
timate (and thereby minimize) the memory required by
each of these stacks.

The code snippet below shows the complete structure
of a TOSThreads TCB. Below, we describe each of the
included fields in more detail:

```c
struct thread {
    thread_id_t thread_id;
    init_block_t* init_block;
    struct thread* next_thread;
    uint8_t mutex_count;    // mutex_count
    uint8_t state;          // thread_state
    thread_regs_t regs;     // thread_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 dynami-
cally loaded onto a mote. As §3.4 describes, whenever
the system dynamically loads a TOSThreads applica-
tion, the threads it creates must receive all the state as-
associated with its global variables. An initialization block
structure stores these global variables and init_block
points to this structure.

`next_thread`: TOSThreads uses thread queues to keep
track of threads waiting to run. These queues are im-
plemented 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 form a queue just
before they wake up. This approach conserves 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 (INAC-
TIVE, 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 struc-
ture the argument points to is not deallocated before
the thread’s start function executes. These semantics
are similar to those that Unix pthreads define.

`joinedOnMe`: This field stores a bitmap of the thread
ids for all threads joined on the current thread through
a join() system call. When the current thread termi-
nates, this bitmap is traversed, 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 in-
terrupted.

`syscall`: This field contains a pointer to a structure
used when making system calls into a TOSThreads kernel.
This structure is readable by both a system call
wrapper implementation and the TinyOS kernel thread.
Following section explains how this structure is used.

3.3 Blocking API

TOSThreads implements blocking system calls by wrap-
ning existing TinyOS services inside blocking APIs. These
wrappers are responsible for maintaining state across
the non-blocking split-phase operations associated with the underlying TinyOS services. They also transfer control to the TinyOS thread whenever a user thread invokes 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 API’s (potentially also written in C) can be implemented as well (e.g. the Tenet API discussed in §4.6).

A user thread initiates a system call by calling a function in one of the blocking API wrappers. This function creates a local instance of a system call block (SCB) structure which contains: a unique syscall_id associated with the system call; a pointer to the thread invoking the call; a pointer to the function that TinyOS should call once it assumes control; and the set of parameters this function should receive. The SCB is used to exchange data with the TinyOS thread.

All variables associated with a system call (i.e., the pointer to the SCB and the parameters passed to the system call itself) can be allocated on the local stack of the calling 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. These variables remain on the local thread’s stack throughout the duration of the system call and can therefore be accessed as necessary.

As discussed in §2, 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, there can only be one outstanding system call at any given time. Thus, only one TinyOS task is necessary to perform the application system calls. The body of this task simply invokes the function the system_call_block points to. This is in contrast to existing threads packages which must maintain system call queues to track the system calls that each thread makes. Figure 2 provides a visual representation of the TOSThreads approach.

### 3.4 Dynamic Linker and Loader

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. In more detail, an applications is compiled offline into a customized loadable binary format we developed called MicroExe [23]. 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 and the linking and loading process.

#### 3.4.1 MicroExe

Binary formats for dynamic linking and loading on general purpose OSs are inefficient for memory-constrained mote platforms. 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 [22], 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. Contiki, another mote operating system, proposed Compact ELF (CELF) to reduce the overhead of ELF with 8 and 16-bit datatypes [5]. While CELF also reduces binary size, the MicroExe format we propose is customized for the more restrictive environment that TOSThreads operates in. Specifically, the ESB platform used in [5] has a 64KB byte-level external EEPROM that the loader uses to store binary images during the linking and loading phase. In contrast, the Tmote Sky does not have this hardware feature. Therefore, TinyLD needs to perform these tasks inline and MicroExe needs to allow for the serial processing of the application binary.

![Figure 2: 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.](image-url)
The MicroExe file format is designed specifically for loadable binaries in TinyOS. It uses 16-bit addresses, compacts the representation of symbol names, and uses chained references [17] techniques to reduce its size. Although it is optimized for the MSP430 platform, its design principles are equally applicable to other microcontrollers. 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.

There are four sections in a MicroExe file (Figure 3). The initial metadata section provides information necessary to decode and patch the rest of the file, such as the sizes of all subsequent sections. The patch table contains information about the type and location of each unresolved address reference in the machine code. To minimize the footprint of this section, MicroExe employs a common compiler technique called chained references [17] to create a chain of linked lists for all the references to the same unresolved symbol. Thus, each entry in the patch table requires only two values: the symbol itself and a pointer to the first chained reference of that symbol. Finally, the initialization table describes how global variables should be allocated, while the code section contains the application’s machine code. In the interest of space we refrain from providing a more detailed description of the MicroExe format. Instead, we point interested readers to the associated MicroExe technical report [23].

### 3.4.2 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 space for all global variables it defines, patches references to local variables, and loads the machine code into the mote’s flash ROM. These steps are conceptually straightforward, and all information required for them is encoded in the MicroExe file itself.

Once all of the linking and loading steps are complete, TinyLD invokes TOSThreads to spawn a new root thread and begin 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 §3.2. 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.

Since child 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 new synchronization primitive, called the 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.

### 4. EVALUATION

In §1 we listed four requirements for TOSThreads: to provide a fully preemptive application-level threads abstraction, be minimally invasive to the existing TinyOS runtime, support a flexible user/kernel boundary, and enable dynamic linking and loading of applications at runtime. Next, 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 as well as MicroExe through code size, both in terms of bytes and lines of code. Finally, we evaluate TOSThreads' ex-
pressiveness 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 4 MHz 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.

### 4.1 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.

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.

### 4.2 Energy Analysis

To measure the impact that TOSThreads has 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 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 pe-

---

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Number of cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Start Static Thread</td>
<td>283</td>
</tr>
<tr>
<td>Start Dynamic Thread</td>
<td>679 + malloc()</td>
</tr>
<tr>
<td>Interrupt Thread</td>
<td>100</td>
</tr>
<tr>
<td>Suspend Thread</td>
<td>145</td>
</tr>
<tr>
<td>Wakeup Thread</td>
<td>15</td>
</tr>
<tr>
<td>Static Thread Cleanup</td>
<td>229</td>
</tr>
<tr>
<td>Dynamic Thread Cleanup</td>
<td>123</td>
</tr>
<tr>
<td>Restore Next Thread</td>
<td>85</td>
</tr>
<tr>
<td>Context Switch</td>
<td>184</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>Number of cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutex Init</td>
<td>13</td>
</tr>
<tr>
<td>Mutex Lock</td>
<td>17</td>
</tr>
<tr>
<td>Mutex Unlock</td>
<td>71</td>
</tr>
<tr>
<td>Barrier Reset</td>
<td>13</td>
</tr>
<tr>
<td>Barrier Block</td>
<td>41</td>
</tr>
<tr>
<td>Barrier Block with Wakeup</td>
<td>6 + 302 × num_waiting</td>
</tr>
<tr>
<td>Condvar Init</td>
<td>8</td>
</tr>
<tr>
<td>Condvar Wait</td>
<td>30</td>
</tr>
<tr>
<td>Condvar Signal Next</td>
<td>252</td>
</tr>
<tr>
<td>Condvar Signal All</td>
<td>314 × num_waiting</td>
</tr>
<tr>
<td>Refcount Init</td>
<td>12</td>
</tr>
<tr>
<td>Refcount Wait On Value</td>
<td>39</td>
</tr>
<tr>
<td>Refcount Increment</td>
<td>11</td>
</tr>
<tr>
<td>Refcount Decrement</td>
<td>11</td>
</tr>
<tr>
<td>Refcount Inc/Dec with Wakeup</td>
<td>11 + 320 × num_waiting</td>
</tr>
<tr>
<td>Join Block</td>
<td>74</td>
</tr>
<tr>
<td>Join Wakeup</td>
<td>74 + 326 × num_waiting</td>
</tr>
</tbody>
</table>

**Table 3: Overhead of invoking the system calls that the standard TOSThreads C API provides.**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Marginal</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sleep</td>
<td>371</td>
<td>715</td>
</tr>
<tr>
<td>StdControl Start</td>
<td>466</td>
<td>810</td>
</tr>
<tr>
<td>StdControl Stop</td>
<td>466</td>
<td>810</td>
</tr>
<tr>
<td>AM Send</td>
<td>390</td>
<td>734</td>
</tr>
<tr>
<td>AM Receive</td>
<td>912</td>
<td>1256</td>
</tr>
<tr>
<td>Sensation</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sensor Read</td>
<td>277</td>
<td>621</td>
</tr>
<tr>
<td>ADC Read</td>
<td>477</td>
<td>872</td>
</tr>
<tr>
<td>Log Sync</td>
<td>466</td>
<td>810</td>
</tr>
<tr>
<td>Log Seek</td>
<td>479</td>
<td>820</td>
</tr>
<tr>
<td>Log Read</td>
<td>491</td>
<td>835</td>
</tr>
<tr>
<td>Log Append</td>
<td>500</td>
<td>844</td>
</tr>
<tr>
<td>Log Erase</td>
<td>468</td>
<td>812</td>
</tr>
<tr>
<td>Block Sync</td>
<td>468</td>
<td>812</td>
</tr>
<tr>
<td>Block Read</td>
<td>495</td>
<td>839</td>
</tr>
<tr>
<td>Block Write</td>
<td>495</td>
<td>839</td>
</tr>
<tr>
<td>Block CRC</td>
<td>506</td>
<td>850</td>
</tr>
</tbody>
</table>

To 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 pe-
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 \( (\text{log\_cost}) \). 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 \( (\text{send\_cost}) \).

As measured in [16], 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:

\[
E_{\text{log\_cost}} = \frac{(\text{log\_cost} \cdot 1.92\,mA)}{4\,MHz} = 2.87\mu\text{As}
\]

\[
E_{\text{send\_cost}} = \frac{(\text{send\_cost} \cdot 1.92\,mA)}{4\,MHz} = 132.23\mu\text{As}
\]

Using an analysis similar to the one in [16], we calculate the total lifetime of this application with different logging periods\(^1\). 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 min sampling interval). Figure 4 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%.

### 4.3 Supporting Long-Running Computations

We evaluate the ability of TOSTThreads 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 TOSTThreads. 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 TOSTThreads 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. TOSTThreads removes this limitation.

### 4.4 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 application 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 §4.2, 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 \( (\text{Code}) \), the total number of relocations \( (U_{\text{Reloc}}) \), the total number of allocations \( (U_{\text{Alloc}}) \), and the total number of initialized global variables \( (U_{\text{Init}}) \). Since MicroExe stores patched

---

\(^{1}\)This analysis assumes that the mote is powered by two AA batteries with an approximate capacity of 2,700 mAh \( (9.72 \cdot 10^3\mu\text{As}) \).
addresses as chained references, \( U_{Reloc} \) is actually equal to the number of unique symbols in the program. The size of a MicroExe file (cf. Fig.3) 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 5 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 \( \mu \)sec on the Tmote Sky). Second, the loading time depends on both the the number of unique symbols in the binary and the number of addresses that TinyLD must patch. This is because TinyLD implements an interative 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 applications. The graph at the bottom of Figure 5 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).

4.5 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 6 and 7 indicate, the TOSThreads versions are more compact in terms of both application code size and lines of code. We gathered binary code sizes by running \texttt{msp430-objdump} and manually counting the number of bytes in the application-specific portion of the binary. We gathered binary code sizes by running \texttt{msp430-objdump} and manually counting the number of bytes in the application-specific portion of the binary. We gathered binary code sizes by running \texttt{msp430-objdump} and manually counting the number of bytes in the application-specific portion of the binary. We gathered binary code sizes by running \texttt{msp430-objdump} and manually counting the number of bytes in the application-specific portion of the binary. We gathered binary code sizes by running \texttt{msp430-objdump} and manually counting the number of bytes in the application-specific portion of the binary. We gathered binary code sizes by running \texttt{msp430-objdump} and manually counting the number of bytes in the application-specific portion of the binary.

Finally, we present a breakdown of the binary code size and RAM usage of a complete TOSThreads kernel compiled together with TinyLD (Figure 8). The kernel used implements the TOSThreads standard C API.
4.6 Tenet

We have re-implemented the Tenet API using TOSThreads. Tenet applications specify tasks as linear data-flow programs consisting of a sequence of tasklets\(^2\). Each tasklet is implemented as a core TinyOS component, providing a specific TinyOS service. For example, an application that wants to be notified when the temperature at any mote exceeds 50°F would write the following task:

\[
\text{Repeat}(1000\text{ms}) \rightarrow \text{Sample}(ADC1,T)\rightarrow \text{LEQ}(A,T,50) \rightarrow \text{DeleteDataIf}(A) \rightarrow \text{Send}()
\]

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 9). 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 [19], and several other custom TinyOS components for accessing sensors and timers.

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:

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

Tenet-C spawns one thread to service each Tenet task and replaces Tenet’s original task scheduler with the TOSThreads thread scheduler. It uses TinyLD to dynamically link and load application binaries (Figure 10). The rest of the original Tenet code runs unmodified but now becomes part of the kernel, running inside the TinyOS thread. 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 benefit of Tenet-C is that the binary code size is smaller than that of the original Tenet, as Figure 11 suggests.

We tested Tenet and Tenet-C 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.

---

\(^2\)Even though Tenet runs on top of TinyOS, Tenet tasks are logically distinct from TinyOS tasks.
4.7 Additional uses of TOSThreads

TOSThreads forms the basis for a high level programming language we have written called Latte—a JavaScript variant for motes. Latte was designed to simplify the writing of efficient WSN applications. Latte programs can either be interpreted within a JavaScript enabled web browser or compiled directly down into C. Running programs in a browser simplifies the early stages of application development and helps to reduce debugging cycles. Programs compiled into C make TOSThreads system calls that are either statically linked against a TinyOS kernel or dynamically loaded onto a running mote using TinyLD. A previous attempt of ours at providing the same end result, TinyJavaScript, was built directly on top of TinyOS without TOSThreads support. Because TinyOS does not support blocking calls, however, TinyJavaScript was forced to expose an event-driven programming interface, increasing the implementation complexity of the compiler and at the same time decreasing the language’s ease of use. Details of both our TinyJavaScript and Latte implementations can be found in [23] and [24] respectively.

TOSThreads has also been successfully used to ease the implementation of a polling-based SD card and GPS driver for the MAMMARK [8] project at UCSC, as well as for upcoming versions of the SPINE body sensor network project from Telecom Italia [14].

5. RELATED WORK

We review prior threading proposals for TinyOS and other sensor network operating systems. While there are many prior and existing thread implementations for mote-class devices, none of them meet all of the four requirements TOSThreads face.

Message passing avoids the synchronization problems that direct kernel traps introduce, allowing TOSThreads to be fully preemptive. For example, TinyMOS [25] which follows a direct trap model, runs TinyOS in a dedicated thread just as TOSThreads does. However, it requires synchronization primitives around core OS abstractions, as the TinyOS concurrency model does not understand preemption outside of interrupts. In contrast, by using a message-passing approach, TOSThreads allows arbitrary concurrency within the kernel, while requiring no changes to TinyOS code except for the interrupt handler post-ambles and the boot sequence.

Putting TinyOS in a separate, high priority thread allows TOSThreads to minimally disrupt existing TinyOS code, unlike TinyThreads [21]. As TinyThreads uses cooperative multitasking based on the TinyOS task scheduler, a single long-running thread can disrupt the task queue and therefore kernel services. In contrast, TOSThreads requires no explicit yields, simplifying programing and preventing errors: users can run multiple infinite loops.

Unlike Protothreads, which do not maintain thread context across blocking calls [7], 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 much more memory than Protothreads, to maintain stacks.

Numerous other concurrency proposals for TinyOS exist, including fibers [26], virtual machine threads [18], and preemptive tasks [3]. None of these approaches allow users to write simple, application level thread-based programs on a TinyOS kernel, because they either limit the number of threads (fibers), are built into a specialized runtime (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 full threads library to applications. While these threads support preemption in principle, none of the current implementations do, instead depending on explicit yield points[4], similar to TinyThreads. Furthermore, it is unclear how full preemption could be safely included without following a model similar to TOSThreads while allowing I/O. Just as TinyOS does with its tasks, Contiki assumes non-preemptive multitasking within its kernel. With preemption, a thread could context switch while in the middle of a kernel call, causing kernel state to be inconsistent and possibly corrupt.

Message passing in operating systems is not new; it is a staple of microkernel designs. 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 re-entrancy 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 L4Linux exhibit slowdowns of 20-150% [12]. Virtual memory is a major cause of this slowdown, and the cost of system calls in multitthreaded operating systems generally. Motes, with their low-power microcontrollers, do not suffer from the major costs of context switches common to high-performance processors. They do not have virtual memory, removing the cost of TLB flushes, nor do they have speculative execution, removing the cost of a pipeline flush.

There have been other proposals for dynamically loading binaries on mote platforms. Like TinyLD, Flex-
Cup allows dynamic loading of TinyOS components [20]. 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. On the other hand, TinyLD does not have these limitations.

Contiki is another mote operating system that supports loadable objects [5]. Contiki uses Compact ELF (CELF) binaries which, like MicroExe binaries, are also compressed versions of ELF binaries. Although both file formats derive from ELF, they are not compatible with it. TinyLD works in more restrictive environments because it does not assume the existence of a byte-level external memory (the ESB platform used in Contiki has a 64KB EEPROM for the loader to store the binary image during the linking and relocating phase).

6. 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 time-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 OS 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.

7. REFERENCES


