SIMPLIFYING CONCURRENT PROGRAMMING IN SENSORNETS WITH THREADING

WILLIAM PATRICK MCCARTNEY

Bachelor of Computer Engineering

Cleveland State University

May, 2005

submitted in partial fulfillment of the requirements for the degree

MASTERS OF SCIENCE IN ELECTRICAL ENGINEERING

at the

CLEVELAND STATE UNIVERSITY

May 2006

This thesis has been approved for the

Department of **ELECTRICAL AND COMPUTER ENGINEERING**

and the College of Graduate Studies by

_____

Thesis Committee Chairperson, Dr. Nigamanth Sridhar

_____

Department/Date

_____

Dr. Chansu Yu

_____

Department/Date

_____

Dr. Dan Simon

_____

Department/Date

To my wife Esther...

ACKNOWLEDGMENTS

I would like to thank all those who gave me the opportunity to complete this thesis. I would like to first thank my advisor, Professor Nigamanth Sridhar, for all the support and encouragement. I would also like to thank the Department of Electrical and Computer Engineering, of which I have had the pleasure being a student the past four years. I want to thank my wife for supporting everything I do. Most of all I would like to thank my children, whose fascination of blinking lights on circuit boards rivals my own.

SIMPLIFYING CONCURRENT PROGRAMMING IN SENSORNETS WITH THREADING

WILLIAM PATRICK MCCARTNEY

ABSTRACT

Wireless sensor networks (sensornets) are deeply embedded, resource constrained, distributed systems. Sensornets are generally developed in an interrupt- (or event-) driven programming model. Writing event-driven programs is hard. Sensornets are generally battery powered, desiring event-driven executions, for power efficiency.

This thesis presents TinyThread - a safe, lightweight threading model which enables sensornet development using procedural programming. TinyThread allows developers to intermix event-driven programming with threaded programming. This frees developers to use whichever programming paradigm is preferred for a specific algorithm. TinyThread operates in the context of TinyOS allowing the concurrency detection mechanism of TinyOS to aid developers. TinyThread includes a stack analysis tool, stacksize which computes all the necessary resource requirements. This cross-platform stack analysis tool enables TinyThread to operate safely across several hardware platforms. TinyThread is completely modular, so only the components required need be included. TinyThread is validated through several experiments testing its power consumption, response time and resource usage.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

Writing distributed system programs is hard. This subject has been heavily researched since the 1970's, and continues to be highly researched today. Even with all the additions of resources in modern computing systems, these problems are still highly researched. A majority of this research is in the domain of fault tolerant, secure, systems. Most of the research in this area concentrates on nodes with hundreds of megabytes of RAM and billions of operations per second. To simplify distributed systems programming, many large steps have been accomplished.

The progress made since the 1970's has openned the doors to some very practical solutions to distributed systems problems. The Internet Protocol (IP) solves the multi-hop routing problem in a clear, and well understood fashion [26]. The Transport Control Protocol (TCP) layers on top of IP and provides reliable transport between any nodes with full re-ordering of messages [27]. TCP also has bounds on transport delays, which can be used to learn of node failures. Even time-synchronization has been trivialized mainly by Network Time Protocol (NTP) and now by IEEE 1588 (which can achieve sub-microsecond accuracy) [9, 22]. There has been a considerable

amount of investigation into group communication middlewares providing reliable, ordered communication between groups of nodes. These contributions solve a vast number of communication and synchronization problems inside of distributed systems.

Nearly all of the aforementioned contributions are currently used on multi-threaded operating systems, most with virtual memory. With a threaded operating system, a developer can pretend that each program or thread gets its own processor. This ability to forget the complexity of managing multiple tasks can significantly reduce the development time. Modern distributed systems can use some type of dynamic memory allocation, which can provide developers another abstraction away from manual resource allocation. Another common component in the nodes of most distributed systems is the memory management unit (MMU). The MMU allows developers to over allocate the memory required for a specific task, and only physically allocate memory as dynamically required. The MMU also can isolate threads or processes into protected memory spaces, which simplifies debugging, since one process cannot corrupt the memory of another process. These additions in operating systems and computer architecture further simplify developing distributed systems.

Even with all these improvements, distributed system development still isn't easy, but the application, security, and reliablity requirements are ever-expanding. Applications are no longer bound by research into classical distributed systems, with some minor exceptions. Distributed systems are used in most industries for everyday tasks. Systems are built with huge metrics that push the mean time between failures beyond imagination. All of these are advancements which have not only simplified distributed system development, but require a significant amount of resources.

A new branch of distributed systems research has recently sprung up in the application of *wireless sensor networks* (*sensornets*). Sensornets provide a much more

resource-constrained type of distributed system. Sensornet nodes generally have a microcontroller onboard with anywhere from 1k to 10k of RAM and 16k to 48k of flash. The resource-constrained nature of sensornets breaks (nearly) all of the aforementioned contributions.

Sensornets are deeply embedded distributed systems. They are first and foremost a distributed network of resource constrained nodes. In most installations, sensornets are battery-powered. This puts a large focus on power consumption. Timing is a also large issue with sensornets, since if one routine blocks the processor for too long, a message could get lost. In summary, sensornets are distributed, resource-and timing-constrained, systems which should be optimized for power efficiency. These challenges drive researchers to developing better tools and programming paradigms.

The challenges provided by sensornets are not as interesting as the new applications constantly being found for sensornets. From border security, to structural monitoring, to guiding firefighters through burning buildings, sensornets are finding applications in a variety of places [2, 16, 20]. All this has been done using extremely difficult programming paradigms and a lack of mature tools. Imagine the applications that could be implemented with better tools, and cleaner programming paradigms.

1.1 The Problem

Multithreading simplifies many programmatic problems. Multithreading/multitasking is by far the most common programming paradigm which even transcends languages. The typical Computer Science undergraduate program teaches procedural programming, achieving multiple simultaneous tasks mainly through threads or processes. In the context of programming methodology, the only difference between a thread and a process is the separation of memory space. Regardless of the language being taught, multithreaded programming is understood by nearly every undergrad-

uate student before they graduate. The popularity of multithreading stems from its distinct ability to separate out simultaneous *tasks* inside of a single system.

Despite its popularity, preemptive threading actually introduces concurrency problems [28]. The simplest example of this problem is two threads sharing a data structure. If one thread is preempted with the data structure in an invalid state, the system may deadlock. The root cause of this problem isn't multithreading; it stems from preemption. These concurrency problems are generally dealt with using semaphores, mutexes, or a variety of synchronization primitives. Even though these primitives fix the concurrency problem, developers must be extremely careful when and where to utilize them, to avoid performance costs and possible deadlocks. These synchronization primitives are well known, and are commonly used throughout several different programming languages.

Implementing threading inside of deeply embedded systems can create three major additional complications. First, memory resources are limited, which is a large problem as each thread requires its own stack. Second, these stacks must be allocated with enough memory so that they cannot overflow, even in the face of multiple interrupts. While the ideas of dynamic stack allocation may appear attractive, statically allocating resources is extremely desirable in an embedded system. The final possible problem is the cost of context-switching. Preemptive multithreading gives each thread a time-slice to run, after which it forces a context switch. Forcing a thread to preempt every finite period of time makes enormous sense on a system which runs at 1GHz in a fully superscalar fashion, but with a microcontroller running at 1MHz the same practice of preempting the current task can severely reduce performance.

While some complexities exist, a multithreaded programming paradigm would be very conveinient for developers to utilize in many applications in the context of sensornets. Developers could learn to program sensor networks in less time, since the

programming paradigm and the reasoning methodology is the same as higher level languages. In contrast, experienced developers could explore much more complex algorithms, without having to worry about a more complex system of programming and reasoning. The core of the problem is not so much implementing multithreading, it is making it accessible and safe in sensornets.

1.2 The Thesis

Multithreading simplifies sensornet development and can be implemented in a safe way. By using static analysis, all required resources can be determined at compile time and concurrency problems can be detected. The resource costs of threading can be understood and controlled. As an example, we present TinyThread.

1.3 The Solution Approach

TinyThread provides safe multithreading for sensornets. TinyThread is a multithreading library for sensornets that safely and efficiently provides multithreading capabilities. All the required resources are allocated at compile time. TinyThread operates in the context of TinyOS, allowing the TinyOS concurrency fault detector to aide developers. TinyThread uses *cooperative threading* as opposed to preemptive threading to simplify the concurrency between local threads.

TinyThread allows developers to write code in the same programming style as languages like C or Java, while still maintaining tolerable costs. The costs, rated by importance, are power consumption, response time, and memory consumption (both program and data memory). TinyThread should optimize for those costs based upon the requirements of sensornets.

TinyThread includes several other components besides simple multithreading. TinyThread also provides access to all levels of the hardware through either existing functions, or expands support by adding Blocking I/O routines which are commonly used in threaded platforms. TinyThread is completely modular, so only the parts required need be compiled. This can reduce the memory footprint(RAM and ROM), as well as improving the performance of the system. Also included are commonly found synchronization primitives which are well known from other languages.

TinyThread is a multithreading library for TinyOS, the de facto standard sensornets research platform [13]. TinyOS is a fully event based operating system written in nesC [11]. TinyOS follows some simple design rules which TinyThread also follows:

- All resources allocated at compile time.
- Analyze for as many problems as possible, prior to execution on a node.
- If it is even statistically rare that something (bad, in particular) can happen, assume it will happen.

TinyThread allows developers to mix event driven programming with threaded programming.

TinyThread implements cooperative multithreading as opposed to preemptive multithreading, making concurrency analysis of TinyOS functional. This also makes many of the synchronization primitives rarely used. As opposed to having all threads contend for the processor at all times, cooperative multithreading allows developers to choose how long their thread should run. This can be difficult if a thread must run a very computationaly expensive operation which resides in another library. But in sensornets, long-running calculations are the exception, not the rule. Also, TinyThread provides an adapted version with preemption support, although it breaks any concurrency detection, making much of development more difficult. Cooperative

multithreading allows for the least amount of transitions possible, since every point of context switching is explicitly specified. This also allows developers to get a clear view of what state the memory will be in when it transitions from one thread to the next.

TinyThread includes a stack analysis tool called *stacksize*, which not only calculates the stack size of the main loop, but also generates a header file with each of the stack size requirements for thread. The stack analysis tool is, in general, new to TinyOS. TinyOS was once been supported by another stack analysis tool, but only for a specific platform. TinyOS's model is to detect as many faults as possible at compile time, but it never automatically analyzes stack size. *stacksize* overcomes that limitation, and allows TinyOS developers to validate that no stack overflow can occur. Also, a cross-platform stack analysis tool means that TinyThread can operate safely across several different hardware platforms that TinyOS supports.

To validate the thesis, several experiments will be explored. These experiments will compare threaded versus unthreaded implementations of the same algorithm. Some of these experiments will measure the power consumption, while others will test response time. Another set of experiments will analyze resource consumption with equivalent implementations of the same algorithm.

Included is a review of the primitives included in TinyThread. Following each primitive is an example application, to show how it simplifies or solves a problem. By reflecting back on other research, much insight into the differences between threaded and event driven programming methods are exposed. Finally, a discussion of the amount of code and its readability, involved in implementing the same algorithm in different design paradigms.

1.4 Contributions

This thesis makes several contributions:

- A multithreading for TinyOS, without any invasive modifications to the base OS.
- A cross-platform stack analysis tool providing a stack usage report, and header generation.
- A library of blocking I/O routines to simplify development of sensornet applications.

1.5 Organization of the Thesis

The thesis is organized in the following way. Chapter 2 contains in depth look at the architecture of TinyThread. Chapter 3 explains the methodology of stack analysis and stack size calculations. Chapter 4 includes the experiments methodology, results and discussion. Chapter 5 reviews similar research. Finally, Chapter 6 contains the conclusions of the results of TinyThread.

CHAPTER II

TINYTHREAD

TinyThread is cooperative multithreading library for TinyOS. It brings the capability of blocking I/O to an event based programming OS. TinyThread internally is written in an event driven paradigm, but developers can utilize TinyThread and its capabilities to write fully multithreaded applications. TinyThread also supports running preemptive threads, but it is disabled by default. Preemptive threads break the concurrency detection of TinyOS, also any regular TinyOS routine must be executed in an atomic statement, since those routines are always expected to run to completion. Preemption is useful for an extremely long calculation, such as running encryption from a C library.

2.1 Background

Since TinyOS and nesC are a base for TinyThread it is important to explain their architectures. nesC is a programming language that was created specifically to support sensornet development [11]. It includes a new object model which is signif-

```
configuration Blink{
}
implementation {
  components SingleTimer, Main, LedsC, BlinkM;
  Main.StdControl -> SingleTimer.StdControl;
  BlinkM.Timer -> SingleTimer.Timer;
  BlinkM.Leds -> LedsC.Leds;
  Main.StdControl -> BlinkM.StdControl;
}
```

Figure 1: nesC Wiring file for Blink application.

icantly different than that of Java or C++. The nesC object model is essentially a way to *wire* together seperate *modules*. These modules can be filled with code more wiring, creating a hierarchical wiring of objects. These modules are not instantiated multiple times if they are wired to multiple times, instead there are ways to automatically generate dispatch tables instead. The wiring points on these modules are called *interfaces*. These interfaces are truly bidirectional. Calls to lower layers are called *commands* and call-backs to upper layers are called *events*. So each module has a set of interfaces it *uses* and another set which it *provides*. An example wiring file can be found in Figure 1.

nesC is implemented as a preprocessor to C. So all of these wires and interfaces get actually replaced with the proper calls. The output of the nesC compiler is a large C file. One key advantage is that events, or callbacks, are replaced with the function call, as opposed to requiring a function pointer. Since the call is actually made, it can possibly be inlined. This is important because it allows extremely light-weight layers to avoid the overhead involved in a function call. nesC also has something similar to threads called *tasks*. Tasks are essentially a way of executing something on the main loop. Tasks always execute to completion, unless an interrupt occurs, after which it will continue to execute. Tasks are extremely lightweight and are extremely fast to execute. One new keyword that nesC adds is the *atomic* statement. atomic is a keyword which disables interrupts in the enclosed section. Also included in nesC is a concurrency analyzer which looks for any data concurrency problems between

interrupts (*async events*), events and tasks. This occurs if data is operated upon in both an async event and the rest of the system. It is important that the data is never inconsistent regardless of when an interrupt can possibly occur. Developers can use atomic statements to protect against data corruption.

TinyOS is a lightweight operating system written in nesC for sensornets [13]. TinyOS is essentially a set of nesC components which implement many commonly required functionalities for sensornets and embedded systems in general. TinyOS includes a network stack for best-effort transport between nodes (AMSTANDARD). It also includes several general purpose components such as software timers, and LED controls. It is important to note how these components interact, using a split-phase operations [11]. This entails one call, usually a command to a lower layer starting some transaction. Later on, an event(callback) is signalled from a lower layer returning either the result or notifying of completion. These allow for extremely efficient code generation, but it forces developers to break apart their algorithms into many event handlers.

2.2 Architecture

TinyThread allows developers to fully intermix the event-driven programming paradigm with the threaded paradigm. This allows for maximum programmability and functionality. This flexibility stems from its architecture.

2.2.1 Software and Hardware Platform Support

TinyThread is implemented as a library to TinyOS, with versions for TinyOS 1.x and the new TinyOS 2.0. At the time of writing TinyOS 2.0 has not been released, although TinyThread has been tested with the latest beta release of TinyOS 2.0. The wiring files differ immensely, as each has version has a different object model.

For instance, generic objects were introduced in TinyOS 2.0 which can actually be instantiated. The TinyOS 2.0 version takes advantage of this, simplifying wiring files. The benefit of the similarities of the interfaces of TinyThread is that it is very simple, almost trivial, to port applications from one version to the other.

TinyThread supports every platform the official branch of TinyOS supports. TinyThread was designed to support the most popular sensornet platforms, namely TelosB (T-Mote), Mica2 and MicaZ. These platforms run using TI MSP430 and ATMEL ATMega processors. Since most other officially supported platforms use the same families of processors, TinyThread supports them as well. There are a handful of research projects which have yielded ports for new platforms using new processors. Some of these ports, namely the platforms using GCC, can be easily ported. At least one of the unofficial ports was made to a PIC16F877, which cannot support threading since it does not have an accessible stack pointer. For all official ports, and many unofficial ports, TinyThread is supported completely.

2.2.2 Interfacing TinyThread

TinyThread supports several different methods of utilizing its interfaces. It allows users to use only the transmit portion of the network from a thread, and the receive may still running an event driven interface. This separation and cleanliness of code allows a variety of algorithms to be implemented in whichever a developer should choose. The top level diagram for TinyThread is shown in Figure 2. Notice how the interfaces at every level are actually exposed to the developer. It should also be known that TinyThread can be completely bypassed, allowing direct access to the underlying TinyOS objects. This ability to mix and match features opens developers to picking and choosing which parts of TinyThread are right for a specific application.

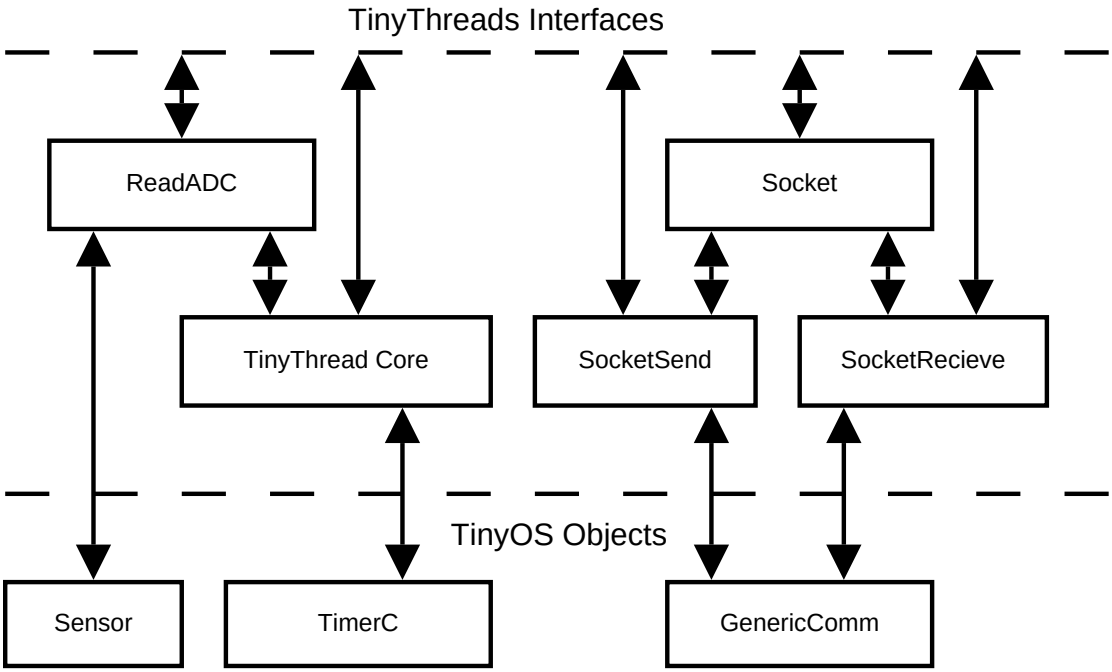


Figure 2: Top Level Diagram for TinyThread.

2.2.3 Stack Swapping

To implement threads, some part of the source code must be written in assembly, since there is no way to save the state of the processor and swap stacks in ANSI C. To facilitate multiple platforms, a separate header must be written for each processor family. The header must contain six macros defining inline assembly routines to do the following:

1. `PREPARE_STACK` - initialized a new thread's stack
2. `PUSH_GPR` - pushes the general purpose registers
3. `PUSH_STATUS` - pushes any status registers
4. `SWAP_STACK_PTR` - swap between two stack pointers
5. `POP_STATUS` - restore the status of the processor

6. `POP_GPR` - restore the general purpose registers

These six macros each play a specific role in swapping from one thread to another. `PREPARE_STACK` operates on a new stack and pushes dummy values into positions, allowing the stack to be switched to later. `PUSH_GPR` pushes more than just the general purpose registers, it actually pushes any registers whose state needs to be saved. This includes any special purpose registers, floating point registers or co-processor registers. `PUSH_STATUS` actually pushes any status registers which may contain: carry out, carry in, overflow, interrupts enabled, etc. It also may push any additional stack pointer register, such as a frame pointer or an argument stack. `SWAP_STACK_PTR` does exactly what it claims, it swaps the two stack pointers. On certain platforms, such as the ATMega series, swapping two stack pointers can be very difficult if the general purpose registers cannot be damaged. This is not a problem, since the general purpose registers are already pushed onto the old stack, and the new stack's copy of its registers will overwrite any modifications. There are also two matching stack popping routines.

Swapping the stacks involves more than simply calling these macros; the order of invocation is incredibly important. The only exception is with `PREPARE_STACK`, which simply modifies some global data without changing the state of the processor. Every time these macros are used, they are always used in the following was shown in Figure 3. Developers who need to implement these basic routines for new processors can safely assume that these routines will always be called in order. If these macros were ever invoked out of order, the system would most likely freeze. One important note on Figure 3 is that the function must not be inlined, since calling the function itself actually places the program counter on the stack, so it does not have to be manually stored.

```
1 void noinline yield(){
2   PUSH_GPR();
3   PUSH_STATUS();
4   SWAP_STACK_PTR();
5   PUSH_STATUS();
6   PUSH_GPR();
7 }
```

Figure 3: Stack swapping routine demonstrating usage of assembly operations.

Implementing a threading system on a modern PC is far simpler than a microcontroller. Besides existing threading libraries, there are functions which exist on many platforms to support implementing user-space threading. For UNIX based systems there are `makecontext` and `swapcontext`, which correspond to `PREPARE_STACK` and the swapping routines. In Microsoft Windows operating systems, there are routines called `GetThreadContext` and `SetThreadContext`. While these routines do not directly correlate to the TinyThread stack routines, they can be built upon to form identical routines. Since all of the stack swapping routines exist, implementing a cooperative user-space threading library is simple.

2.2.4 Scheduler

The TinyThread scheduler operates inside of a TinyOS task. Essentially, it gets executed as any other task, except that it may swap to another context. As far as how this changes the way TinyOS executes, it allows TinyOS to continue running in the *system* thread. The TinyThread scheduler will allow one thread to execute until it blocks. After the thread has blocked, the scheduler switches back to the system context. If there are any active threads, then the scheduler will post itself back into the TinyOS scheduler, otherwise it will wait until an event has happened to wake up a thread before posting itself. Each thread operates like a finite state machine (FSM) clocked by specific routines. Figure 4 visually shows the FSM for a single thread. Threads can transition through states of LOCKED, IO, and SLEEPING. All calls out

of the active state involve the thread itself making one of those calls, forcing itself inactive. These routines are called by the routines directly exposed to the user as demonstrated in Section 2.3.

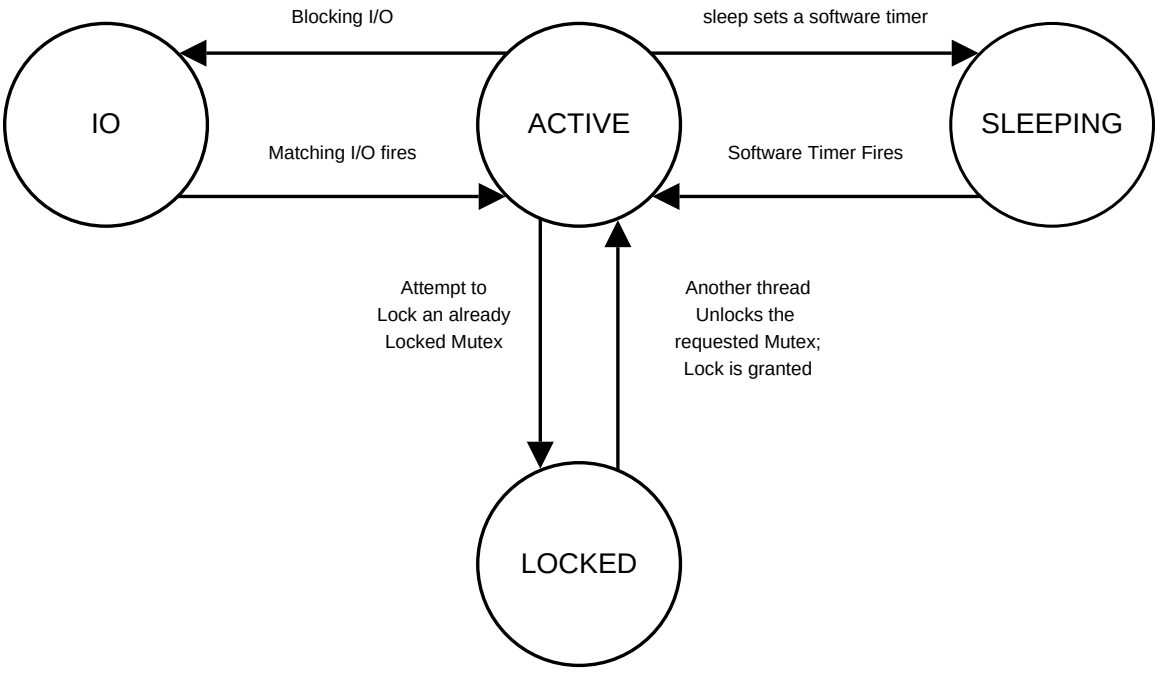


Figure 4: State Diagram for Thread Scheduler.

The finite state machine, shown in Figure 4, is a purely event-driven system with some of the events coming from threads. The blocking I/O routines operate in a very simple manner. When a blocking I/O call is made, the thread begins the I/O transaction and then goes into the IO state. This thread never returns into an active state until some other event forces the thread out of the IO state. The thread enters the LOCKED state when it attempts to lock a mutex which is already locked. The thread will only leave the LOCKED state when another thread that owns the mutex calls `Mutex.unlock`. This state machine is duplicated for each thread.

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

Figure 5: Blocking I/O interfaces provided by TinyThread.

2.3 Programming Primitives

The ability to switch between contexts is not extremely beneficial, but context-switching is the basis for implementing better programming primitives. The benefits of programming with threads compared to a purely event-driven environment is the ability to use Blocking I/O primitives. Embedded systems with only one task can block the processor waiting for a specific I/O. In sensornets this is dangerous; if a processor blocks, a event may be missed.

2.3.1 Blocking I/O Primitives

TinyThread provides several blocking I/O routines for commonly used sensornet activities such as radio and sensor access. These routines block the current context so they must only be called from inside of a thread, not a task nor an event. The `BlockingADC` interface provides the `readADC()` function (Lines 1–3 in Figure 5). When called, this function simply blocks the thread until the data from the A-D controller becomes available. In the very next line of code, the program can actually use the data collected from the sensor.

Oscilloscope

The Oscilloscope application that is distributed as one of the samples with TinyOS is a simple application that samples sensors on a mote, and sends this sampled

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

Figure 6: Implementation of the `Oscilloscope` application using threads. The `Socket Send()` routine is wired to a fully buffered fifo sending queue.

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

Contrast this with our implementation of the same application written using TinyThread's blocking I/O operations, shown in Figure 6. The logic of the program is now much easier to see, and there is no need for manually ripping the function into pieces.

The `Socket` interface (Lines 5–9 in Figure 5) provides blocking operations for sending and receiving messages. The `send()` operation in `Socket` blocks until the message has actually been sent (until the `sendDone()` event is raised in a split-phase

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

Figure 7: Implementation of the `Bounce` application using threads. This example illustrates the blocking receive functionality in `Socket`.

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

Figure 8: Synchronization interfaces provided by TinyThread.

implementation). This means that if a node needs to send a series of messages, they can actually be sent out in a loop rather than ripped apart in several functions. The `receive()` operation simply blocks until there is a message waiting to be processed.

Bounce

This is a very simple application intended to illustrate the use of the blocking receive operation (Figure 7). The application runs on two motes, and two motes continually bounce a message back and forth. When a mote receives a message, it turns on its red LED, and when it sends a message, it turns on its green LED. The interesting piece in this example is on line 7, where the program simply waits for the next message. The thread does not proceed until a message actually arrives.

2.3.2 Synchronization

Mutex

The simplest synchronization primitive that TinyThread provides is a way for threads to acquire mutually exclusive access to some critical section. The `Mutex` interface allows a thread to `lock()` a mutex and enter its critical section. Once it is done executing its critical section, the thread calls `unlock()` to relinquish critical section access to any other thread that may want to use it.

Barrier Synchronization

A *barrier* is a primitive for *rendez vous* synchronization among a set of threads [17]. The `Barrier` interface in the TinyThread library provides two kinds of barrier synchronization. First, it supports pair-wise synchronization between two threads. The thread that arrives at the barrier first calls the `block()` command. This causes this thread to stay in the blocked state until it is woken up by the other thread. When the second thread has also arrived at the barrier, it calls `unBlock()`, which causes both threads to be active again.

Data diffusion protocols are used frequently in sensornets, in a wide variety of ways [14, 15, 19]. Consider a simple Gossip data diffusion protocol in a network. Some node initiates the "gossip", and each node in the network propagates the gossip until the message reaches the edge of the network. When a node receives a gossip message for the first time, it marks the neighbor from where the gossip arrived as its *parent*. When a node does not have any neighbors other than its parent, it sends the message back to its parent. When a node has received acknowledgments from all of its neighbors, it sends the message back to its own parent. In this manner, the initiating node eventually receives acknowledgment that the entire network has seen the message.

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

Figure 9: Implementation of a Gossip diffusing computation through a network. The two threads in the program synchronize using a pair of barriers.

Figure 9 shows the multi-threaded implementation of this protocol. The two threads—`threadIdle` and `threadActive`—are started at the same time. But the second thread blocks until the node enters the `ACTIVE` state (the node has received its first gossip message, and is now active in propagating the message). The former thread, `threadIdle` is also blocked until the first message arrives. Once the first message arrives, and the node transitions into the `ACTIVE` state, `threadIdle` has also arrived at the barrier that `threadActive` is waiting at, and calls `ActiveBarrier.unBlock()` to signal the rendez vous. At this point, both threads are active.

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

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

Rendez Vous-based Message Passing

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

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

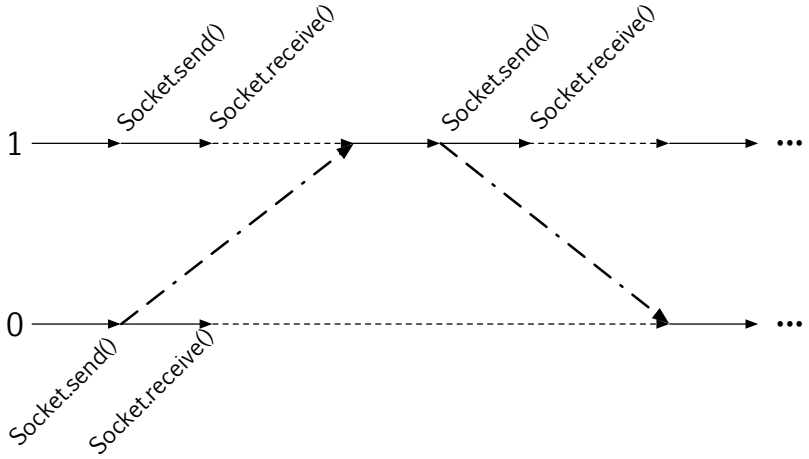


Figure 10: An illustration of *rendez-vous* synchronization in the Bounce application. The dotted lines in each node represents periods where the program is blocked waiting for a message.

CHAPTER III

STACK ANALYSIS

Currently, thread stack size for MANTIS, and many other threaded RTOS's, is done through an educated guess by the developer [3]. TinyOS has no built-in tools to detect stack size, although developers do not have to allocate the stack size manually. There are tools which can calculate the stack requirement for some microcontrollers, but it requires special knowledge of the operating system which is running on the application. To this end, we present a tool which analyzes the stack usage of TinyOS applications for all platforms which TinyThread supports. To analyze stack requirements, it is the worst case scenario stack which must be allocated.

TinyThread uses an augmented call-graph to help analyze the stack usage [29]. Static stack analysis has been explained, and even demonstrated by Regehr, et al [25]. Static stack analysis of TinyOS applications can be done in a very simple way. Simply calculating the stack usage of the main function, added to the maximum stack usage by any task, then adding all the stack usages of all the interrupt handlers. This simple approach to stack estimation, shown in Figure 11, although safe to use, grossly exaggerates the stack requirements.

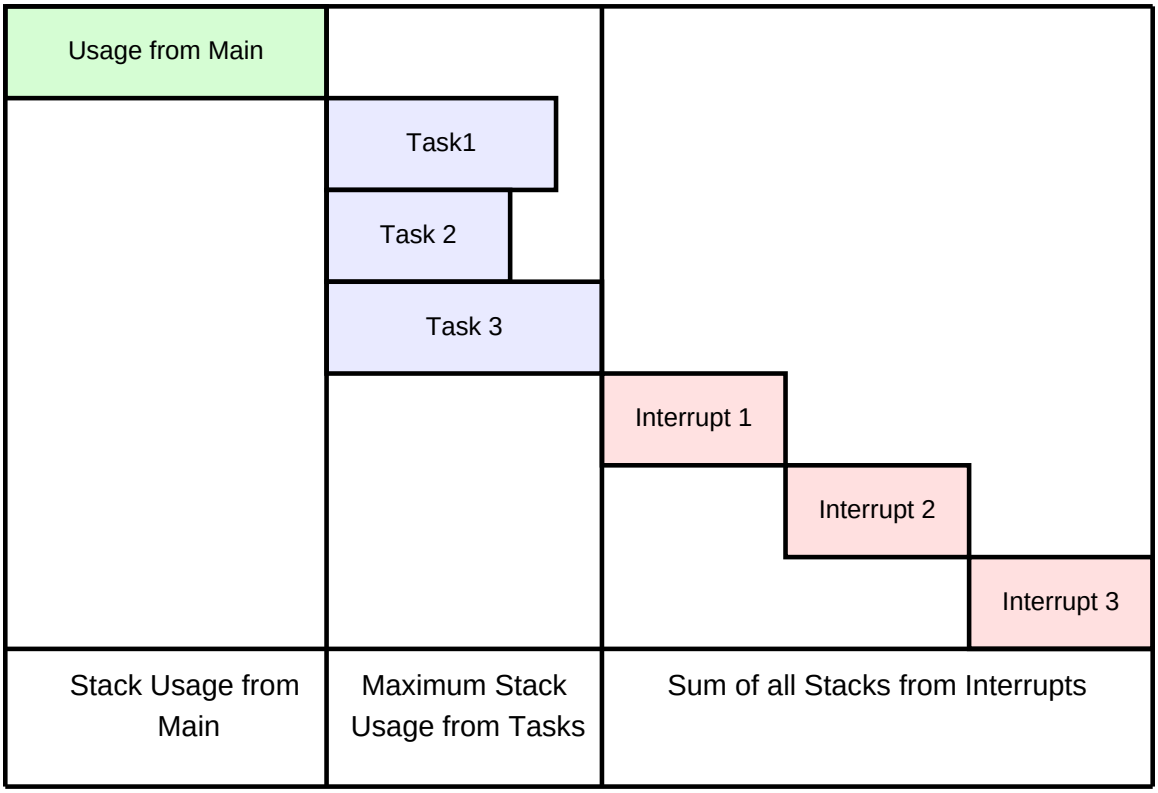


Figure 11: This shows the simple stack usage calculations, adding all the interrupts' stack usages with the maximum task stack usage.

A better way to calculate stack depth is by using a context-sensitive static analysis [25]. The essence of this strategy is to exploit the fact that interrupts are disabled throughout different parts of the application. In TinyOS, this extends far beyond simply inside of events, but it actually continues throughout the interrupt handlers themselves. In terms of the software, GCC supports two different kinds of interrupt handlers: 1. *interrupts* are interrupt handlers that have interrupts enabled and 2. *signals* are interrupt handlers with interrupts disabled. The latter of the two provides an extreme amount of stack savings, since hardware *signals* cannot preempt other hardware *signals* or *interrupts*. For reference, in TinyOS 1.1, in the MSP430 based platforms, all interrupt handlers are actually *signals*, giving excellent stack savings. An overview of the calculations made can be found in Figure 12.

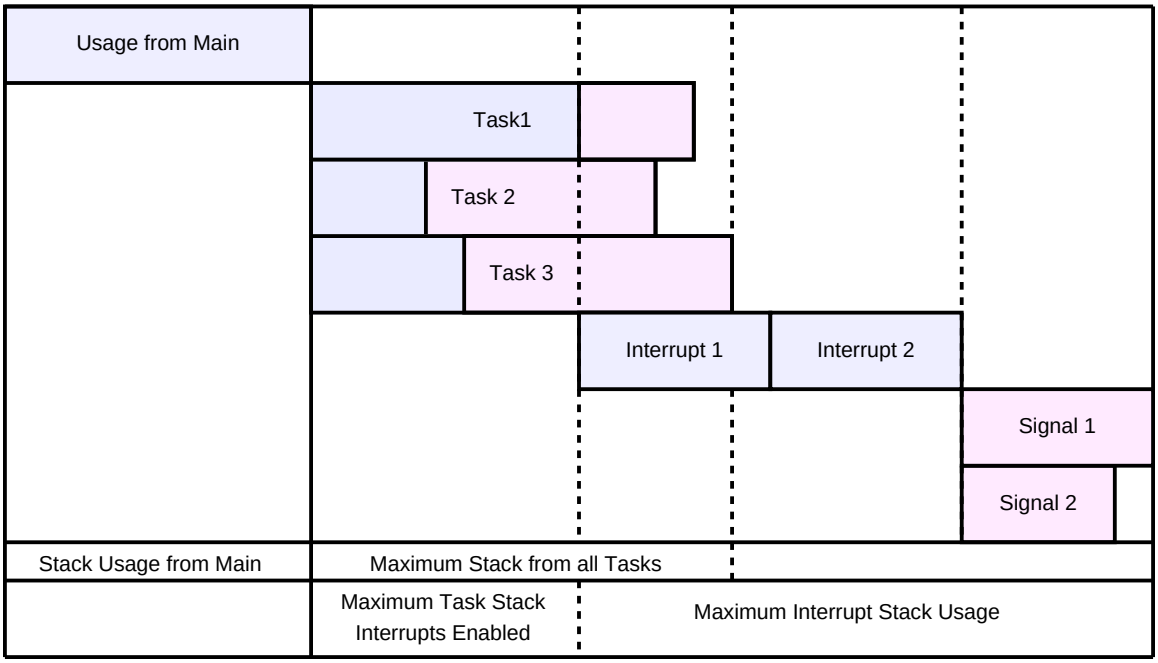


Figure 12: This shows the context sensitive stack usage calculations, adding all the interrupts' stack usages to each other only if and/or where interrupts are enabled.

3.1 stacksize

The stack detection tool provided with TinyThread is called stacksize. stacksize executes a context sensitive stack analysis for several platforms. It can analyze binaries for essentially any ATMega based or MSP430 based platform. Instead of acting as a disassembler, stacksize invokes the disassembler (objdump). Objdump is a standard part of the GNU toolchains. This allows stacksize to be significantly simpler than Stacktool, but it limits it to the GNU compilers. While this appears extremely limiting, the GNU Compiler Collection (GCC) supports dozens of platforms and is well known for its portability. In the scope of TinyOS, this is not a limitation, since it already uses GCC.

stacksize does not support every platform that GCC supports, but it can be extended to do so. Each platform supported requires some platform-specific code.

This glue code binds the assembly code generated by objdump to help form the context-sensitive call graph. Another platform-specific part of the analysis is accessing the interrupt vectors. The pieces of platform specific code are rather small, currently around 100 lines of code per processor family. To add a new platform simply requires filling out the required methods and adding the platform to the command line options. This allows the tool to trivially grow with needs of developers.

stacksize calculates not only the stack required for the main routine, but also for each individual stack. To accomplish stack calculation, stacksize invokes objdump and builds a context-sensitive call graph from the disassembly. It then assumes that every unreferenced function must be a task or a thread. After computing the stack requirements for each individual thread, stacksize generates a header file (stack.h) containing macros for the stack size, and displays the requirements for the `main` function on the console. A typical workflow for stacksizeis shown in Figure 13.

stacksize creates two main outputs, the header file and the main stack usage. The header file is very simple and can be seen in Figure 14. For a developer to use this header, they only have to include it in their application and use the proper nomenclature when declaring the stack for a thread. The nomenclature appends a `_STACKSIZE` to the end of the function name. For instance, to get the stack size for a function named `processing`, the stack size from the header file would be `processing_STACKSIZE`. The output of stacksize displays several pieces of information. It displays the context-insensitive and the context-sensitive stack analysis. The output can be seen in Figure 15.

The output of stacksize is shown in Figure 15. The output is split into two sections. The first section contains only the context insensitive analysis. The second section contains the context sensitive analysis, and displays the final worst-case scenario stack size. One important number displayed below is the “Interrupt overhead

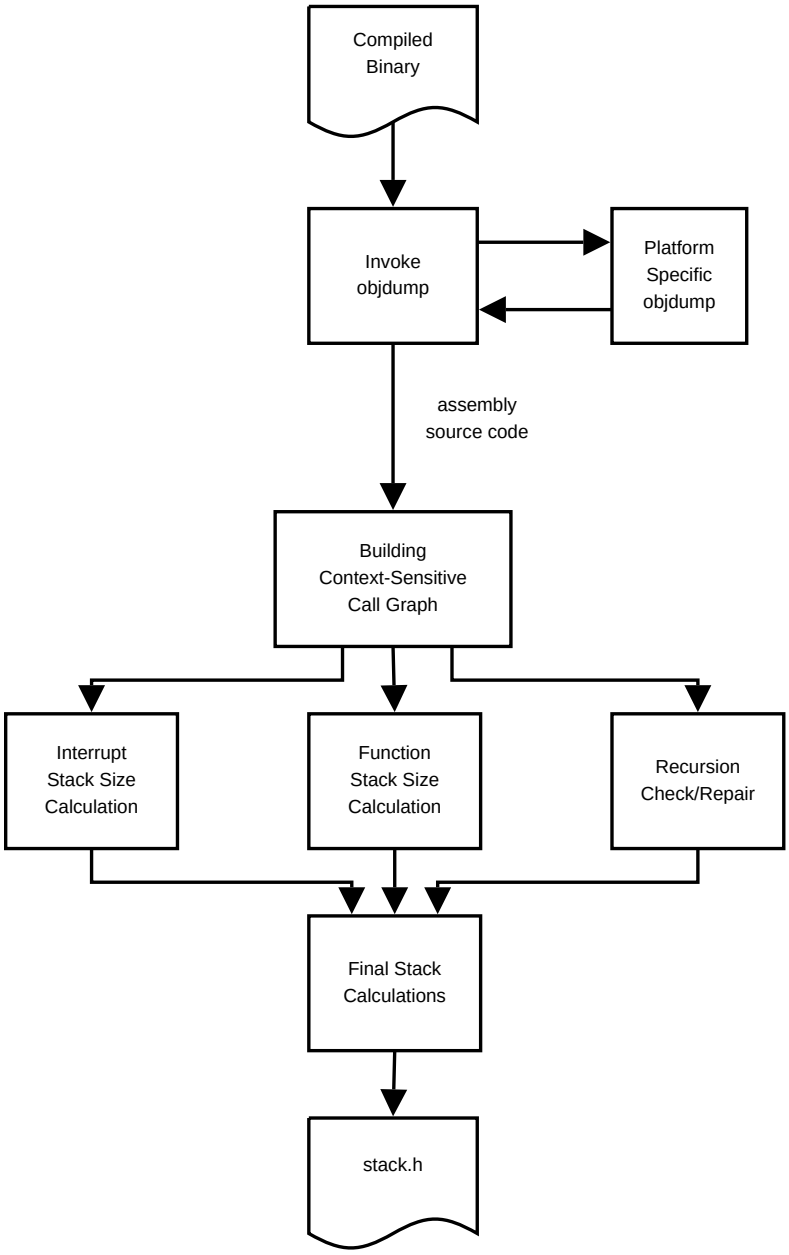


Figure 13: A typical workflow for stacksize.

```
#define  thread_blink_STACKSIZE  84
#define  checkShortTimers_STACKSIZE  80
#define  checkLongTimers_STACKSIZE  78
#define  __ctors_end-0x3a_STACKSIZE  48
#define  thread_task_STACKSIZE  58
#define  thread_wrapper_STACKSIZE  50l
```

Figure 14: An example stack.h header file.

on all stacks". This number is the worst case scenario costs of all nested interrupts.

```
Simple Stack Analysis
Stack due to interrupts =   84
Stack due to tasks =        48
Stack due to Main =         14
                  Total    146

Context Sensitive Interrupt Masking Analysis
Stack due to Tasks                       =   48
Stack due to Main                        =   14
Stack due to Tasks (Interrupts Enabled) =   36
Stack due to Signals(maximum)            =   36
Stack due to Interrupts (calculation)    =    4
Interrupt overhead on all stacks         =   48
                                  Total =   94
```

Figure 15: An example output from stacksize.

stacksize is written in Python, a cross-platform language, and has been tested running inside of Microsoft Windows, Mac OS X and various Linux Distributions. Python is a required component for TinyOS and the runtimes are included in most TinyOS distributions. stacksize was written to specifically support TinyOS. It currently supports all the platforms that TinyOS officially supports (AVR and MSP430 based platforms). There is some preliminary support for ARM processors as well.

To demonstrate stacksize, a series of event driven applications, which were included with the TinyOS distribution, have been analyzed. The stack usages for each platform are shown in Figure 16.

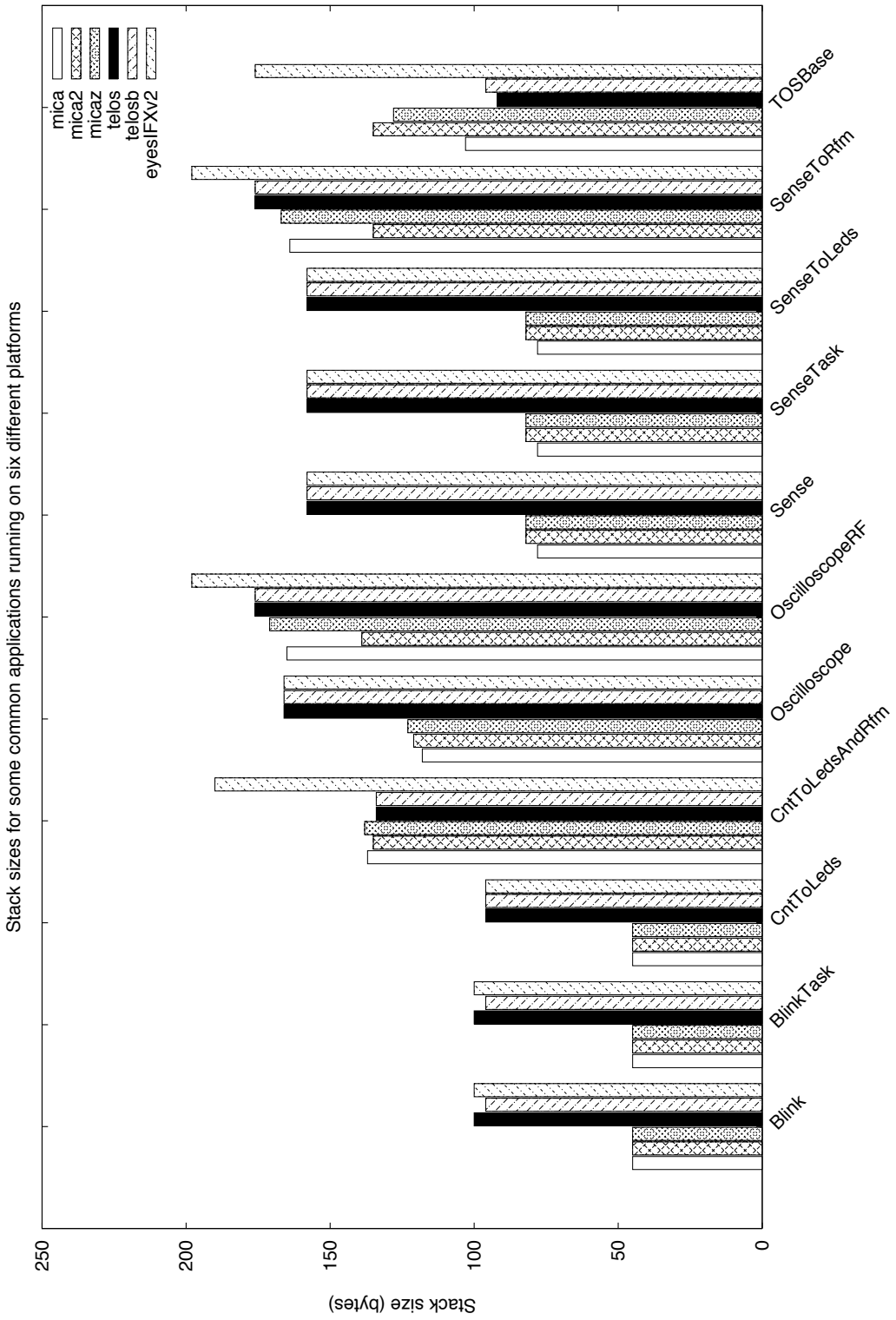


Figure 16: Stack sizes of a few applications from the `tinyos/apps` directory in the TinyOS distribution. The plot shows the result of our stack analysis tool run for the mica, mica2, micaz, telos, telosb, and the eyesIFX2 platforms.

3.2 Limitations

Even though stacksize supports context sensitive stack analysis and generates constants for the thread stack sizes, there are several limitations to its methods and implementation. One problem with stacksize is its inability to monitor each interrupt mask; it only monitors the global interrupt mask. This ignoring of interrupt masks may cause stacksize to overshoot the maximum stack requirements for an application or thread. This limitation is acceptable, since it would require much more specific knowledge of the processor model. This means that there needs to be additional code for each processor, not just the processor family. With dozens and possibly hundreds of models in a specific family, it would be very difficult to write the peripheral specific analysis routines for every peripheral on every processor.

stacksize cannot account for physical implications of different interrupts. stacksize assumes that all interrupts(that run with interrupts enabled) can physically fire simultaneously, and based upon that assumption calculate the stack accordingly. For example, if one interrupt is fired on the rising edge and another on the falling edge of a long pulse, then these interrupts might be physically mutually exclusive. stacksize only calculates without any knowledge of any physical characteristics of an interrupt, since that would make assumptions about the application, and the processor.

Another limitation of stacksize is the inability to support every possible assembly formation. stacksize was written with TinyOS in mind. This means there are several possible constructs which are not supported. For instance, pushing onto the stack in a loop is impossible to do in ANSI C, and therefore is not supported by stacksize. This limitation does not end there. There are several constructs in C which are not supported by stacksize. Currently, function pointers are not taken into account, and will definitely affect stack usage. Recursion is supported, but only if the developer adds a command line argument with the number of recursions it will make.

Without an upper bound on a recursive routine, the stack size cannot be calculated. This includes indirectly recursive routines. If the processor supports reentrant interrupt handlers, the stack size generated by stacksize may be wrong. The analysis only fails if the interrupt can actually fire fast enough to cause it to re-enter prior to the handler finishing up. Another unsupported feature of ANSI C is use of the function `alloca`. `alloca` literally pushes the stack pointer a variable amount, in fact, it could be a parameter of the function.

Generating a header after analyzing a compiled binary has a distinct side effect, the source code must be recompiled again. Every time an application is changed it must go through a much longer compilation process. It must be recompiled once, execute stacksize to generate the header, and then recompiled again using the header. This process is not limited only to application changes, but also to platform switching. This at least doubles all compilation times.

CHAPTER IV

EXPERIMENTS

Several experiments were conducted to measure the performance and costs of TinyThread. These experiments can be broken up into several categories. First type of experiment will measure the power consumption. The next experiment will measure response time through the network stack. The final set of experiments measures the resource consumption: program, memory and stack consumption.

We ran our experiments in our lab on Tmote Sky and micaZ motes. We also tested TinyThread on mica2 and mica2dot motes, although all the measurements presented here are from Tmote and micaZ motes. Applications that had Java interfaces ran on a Pentium IV PC with a 3.2 GHz processor with 512 MB of RAM. The power measurements we present here were recorded using a Tektronics oscilloscope. Our multihop experiments were run on a heterogeneous testbed consisting of 7 Tmote Sky motes and 8 micaZ motes (Figure 17).

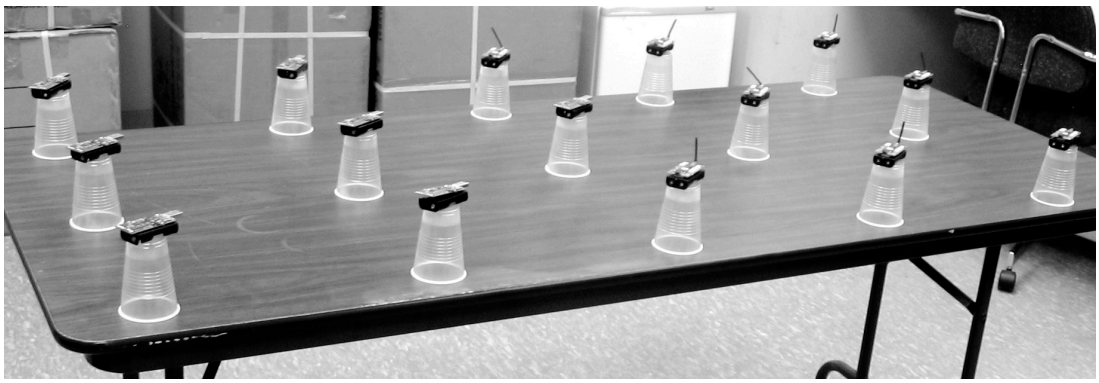
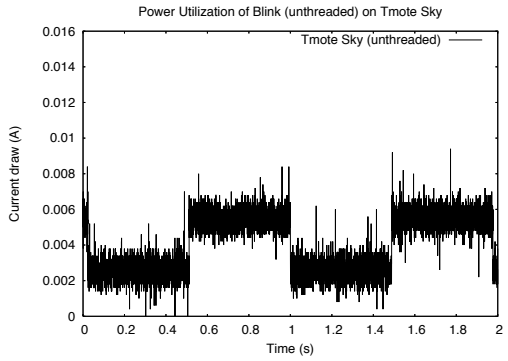


Figure 17: Experimental setup: Our multihop network experiments were run on this heterogeneous testbed consisting of 7 Tmote Sky motes and 8 micaZ motes placed in a 3 x 5 grid.

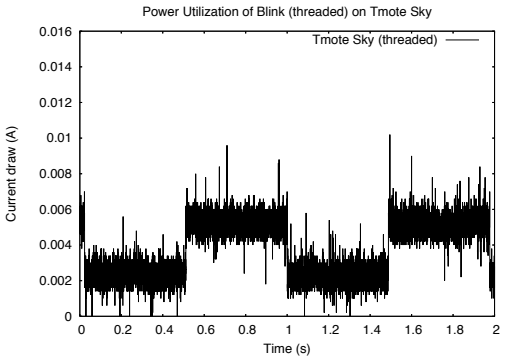
4.1 Power Consumption

Using TinyThread to develop applications *appears* to fundamentally change the way applications execute on the mote. The presence of blocking operations for using sensors and the radio seems to suggest that there may some inefficiency in the amount of power utilized by the mote. This is not the case. The TinyThread scheduler is efficient in terms of power utilization, and in almost all cases, the power draw of a program written using TinyThread is exactly the same as the same program written using regular TinyOS tasks and events. TinyThread is benchmarked against equivalent event driven applications. To measure the power consumed, a 1 Ohm resistor was placed in series with the supply and a digital oscilloscope captured the voltage. Since the voltage accross a 1 Ohm resistor is equivalent to current, all of the following plots are in mA although it was actually measured in volts.

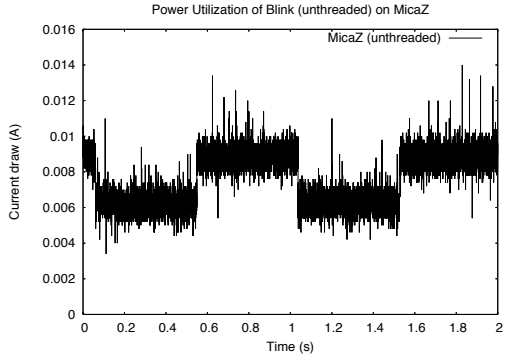
We ran power utilization tests on several simple applications. The first one is the most simple application of all: Blink. Figure 18 is a comparison of the current draw in a Tmote Sky mote and a micaZ mote both running two versions of Blink. As Figures 18(a) and 18(b) show, there is no change in the amount of current drawn



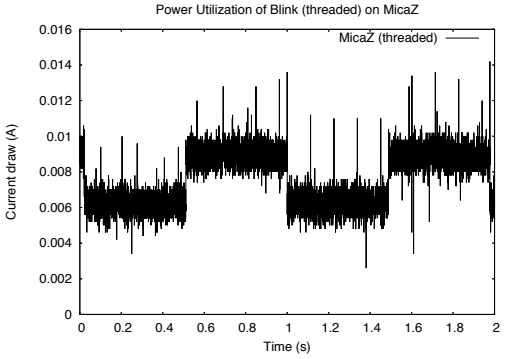
(a) Power draw of a Tmote Sky running Blink (unthreaded).



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

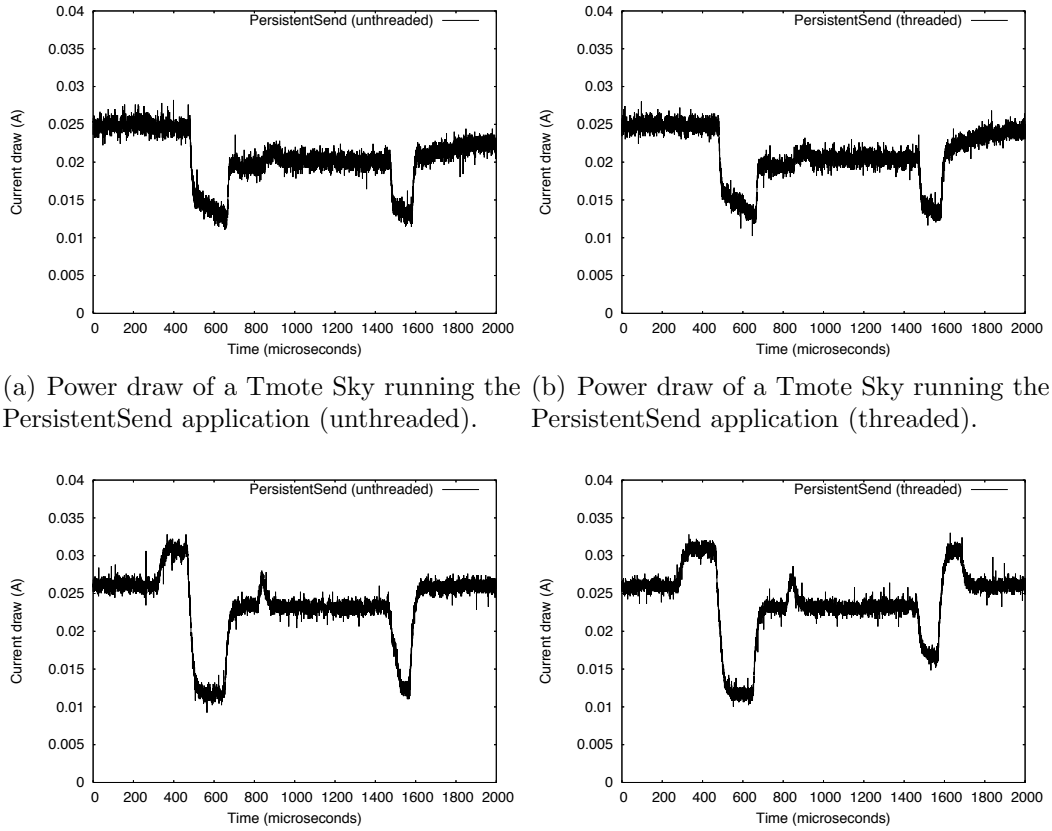


(c) Power draw of a MicaZ running Blink (unthreaded).



(d) Power draw of a MicaZ running Blink (threaded).

Figure 18: Comparison of power utilization of the Blink application running on a Tmote Sky and a MicaZ mote. The figure on the left shows the current draw of the mote running the application without TinyThread, and the plot on the right is with TinyThread.



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

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

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

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

Figure 19: Comparison of power draw on a Tmote Sky and a micaZ running the unthreaded and threaded versions of the PersistentSend application. This application keeps sending out a message on the radio every 30 milliseconds.

in the mote as a result of using TinyThread. The same is true for the micaZ mote as well[1].

These results are consistent with our description of the TinyThread scheduler (Section 2.2.4). Recall that the entire TinyThread scheduler itself is a regular TinyOS task. It gets posted at startup time, and after that, it only gets posted when there is at least one *active* thread waiting to execute. If all the threads in the program are inactive, the TinyThread scheduler never gets posted, and therefore does not cause any extra power drain. This is the key insight behind why the power utilization of a

[1]The current draw in the micaZ is in general worse than the Tmote Sky.

program is the same, regardless of whether TinyThread is used or not.

The second set of experiments related to power utilization we ran were to test the effect of using the radio. We wrote a simple application (called PersistentSend) that sent out a message on the radio every 30 milliseconds. The remaining time, the mote had its radio on in receive mode. Figure 19 shows the power utilization of this application running on a Tmote Sky. The radio's power is at its default level. At this level, the radio on this mote, the Chipcon CC2420 is supposed to consume 19.4 mA of current in the receive mode, and 17.4 mA of current while transmitting [5, 23]. This is consistent with our results. In the plot in Figure 19(a), we measure the power consumption of the unthreaded version of PersistentSend application. In the left end of the PersistentSend plot, the mote has the radio turned on in receive mode. About 400μs into the reading, the mote switches the radio from receive mode to transmit mode, and then begins to transmit. The transmission lasts for about 800μs, after which the radio is switched back to the receive mode. The plot matches well with the values listed in the radio's data sheet. Figure 19(b), which is the threaded implementation of PersistentSend, displays a power profile that is identical to that of the regular TinyOS implementation using tasks and events.

Figures 19(c) and 19(d) show the power profiles of the same PersistentSend application running on a micaz mote. The power profiles in this case are similar, although not identical, as with the Tmote Sky. The extra power consumed is the processor performing the stack swap operations, and getting ready to execute the TinyThread scheduler. Just before the radio is switched to the Tx mode, the figure for the threaded application shows roughly an additional 50 μs duration where the current draw is at .03 mA. The same duration of increased power usage is seen at the other end (once the radio is done sending, and is being switched back to the Rx mode).

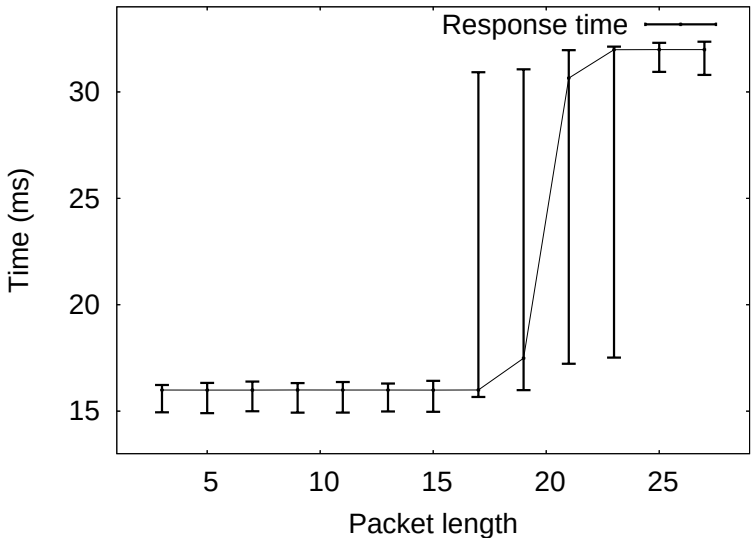
4.2 Response Time

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

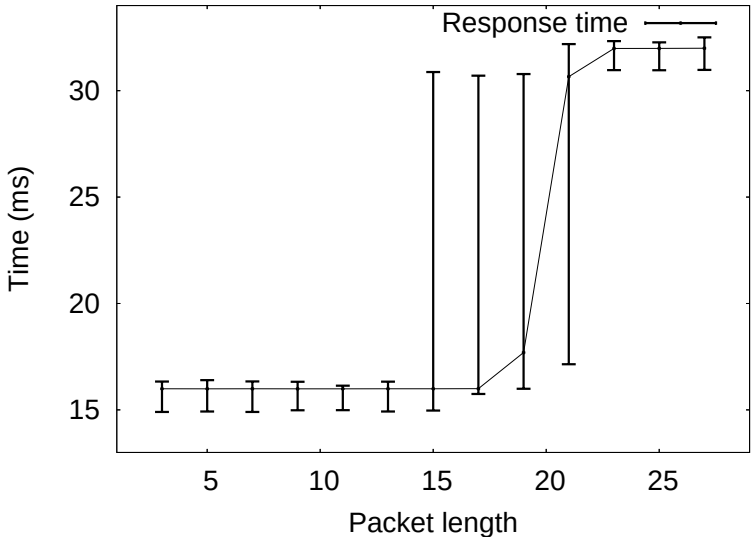
On the MSP430 processor, the stack swapping operation amounts to 34 instructions per swap. This means a total overhead of 68 instructions. On the AVR ATMega 128 processor, the overhead is higher, and is 66 instructions per swap (total of 132 instructions per scheduler cycle). In addition to this finite overhead, when the TinyThread scheduler begins to execute, it has to walk through its list of threads to see which of those is *active*. This will involve a few additional instructions.

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

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



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



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

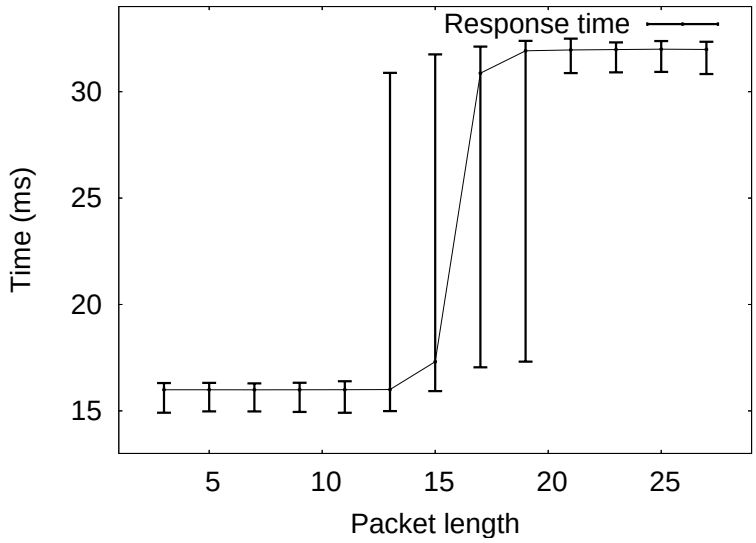
Figure 20: Comparison of round-trip message times from a PC to a mote and back. The response time is measured on the PC.

for each message. The time measurements on the PC were made using the actual processor clock via the Java Native Interface; this is a more accurate measure than using Java's time measurements.

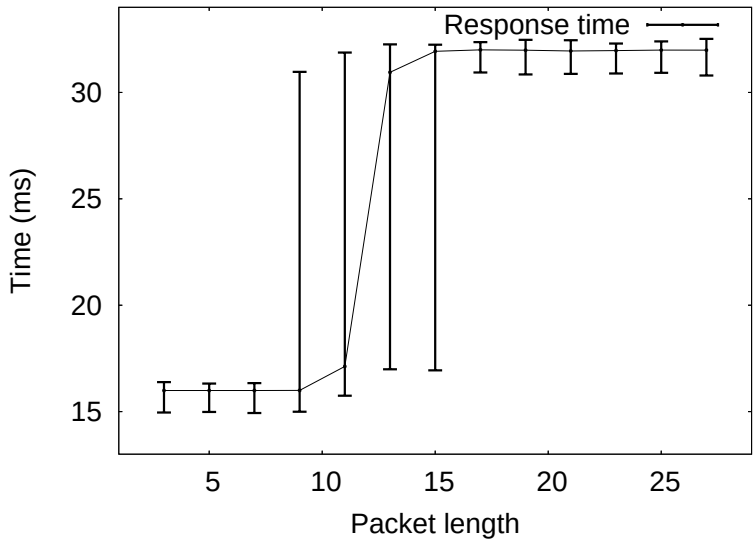
Figure 20(a) and 20(b) show the response times measured over a range of message sizes (3 bytes – 27 bytes). We show the median response time, along with the deviation over the 100 samples. The response times for the threaded version are similar to those of the unthreaded version, but it is possible to see the overhead caused by the thread library.

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

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



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



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

Figure 21: Comparison of round-trip message times from a PC to a mote computing a digital filter. The calculation is a long-running operation that is constantly active. The response time is measured on the PC.

just like in the previous experiment.

The results from this experiment are shown in Figure 21. In this case, the degradation in response time in the threaded version of the program is more perceptible. This is again consistent with our analytical prediction.

4.3 Resource Usage

4.3.1 Thread Driven Applications

To compare the stack and memory consumption of TinyThread with another multithreading OS, stacksize was run on MANTIS. While it would be much more beneficial to survey several operating systems, commercial OSes generally include a license which does not permit publishing any benchmarked results. To understand the results, the limitations of stacksize must be understood from Section 3.2. Since MANTIS is a fully multithreaded operating system, it can benefit from using stacksize.

To compare the results, the MANTIS application tested is their Blink application. In contrast to the TinyOS Blink, MANTIS Blink actually runs all three LEDs, one in each thread. This application actually has three threads. So each thread must allocate its own stack, along with enough for the overhead of all possibly nested interrupts. Since most of the interrupts in MANTIS are true interrupts, and not signals, the theoretical stack consumption is enormous. This problem is exacerbated by fairly few interrupt handlers which are split level interrupt handlers(top level and bottom level). TinyOS is a significantly smaller system for a variety of reasons. Instead of attempting to compare unlike systems, it is better to compare interrupt overhead; the amount of extra stack space each thread must allocate for the use of interrupts. The interrupt overhead for MANTIS 0.9.5 on a T-Mote is 224 bytes. This leads to the minimum stack size of 272 bytes (including overhead from stack swapping). This is

enormous compared to TinyOS which has a stack overhead of 28/34 bytes for Blink on the Telos/Mica platforms. The stack overhead for the Mica style motes increases significantly to 110 bytes when the radio is enabled, but it is still less than half of the overhead due to MANTIS. The extra memory saved is multiplied by the number of threads in the system.

4.3.2 TinyThread Resource Usage

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

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

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

There are four applications to be tested against: *Blink*, *Bounce*, *Filter*, and

$$StackUsage + DataSpaceUsage = TotalRAMUsage$$

Figure 22: Total RAM usage calculation.

SimpleComm. All four of these applications use only one thread. *Blink* turns an LED on and off. *Bounce* passes a message back and forth between two nodes. *Filter* runs a 512 tap FIR filter, and *SimpleComm* was used to test the response time of messages accross the serial port, through the network stack.

TinyThread was compiled several times using all of the combinations of the options, for each of the applications, in order to calculate the cost of each option. Typically, to calculate the optimal stack size, turning off all options which are not required will minimize your RAM and ROM usage while improving your performance. The complete results of compilations organized per platform can be found in Appendix A.

To summarize the resource analysis of TinyThread, Table I compares only the final optimization of the TinyThread versions and the event driven version. This table uses only the ROM usage and the total RAM usage. The total RAM usage is not generated from one succinct source, it is derived from the stack usage of the application and RAM usage of the application as shown in Figure 22. Data space is generally allocated at the lowest available memory location, while the stack generally grows from the highest memory address down. A stack overflow occurs when the top of data memory is about the lowest used stack position. For TinyThread the individual thread stacks are allocated in the data space already.

4.4 Discussion

TinyThread can be compared to several different development models, both threaded and event driven. It has advantages over both types of development models,

		Threaded Version		Event Driven Version		Difference	
Platform	Application	RAM	ROM	RAM	ROM	RAM	ROM
Telosb	Blink	188	3226	136	2610	52	616
Telosb	Bounce	651	12778	511	11780	140	998
Telosb	Filter	2815	12874	2549	11840	226	1034
Telosb	SimpleComm	737	12716	478	12802	259	-86
Mica2	Blink	210	2534	94	1610	116	924
Mica2	Bounce	876	12430	581	10834	295	1596
Mica2	Filter	3132	12690	2642	11088	490	1602
Mica2	SimpleComm	737	12716	545	11326	192	1390
MicaZ	Blink	210	2564	94	1630	116	934
MicaZ	Bounce	884	11784	528	10010	356	1774
MicaZ	Filter	3173	12016	2589	10258	590	1758
MicaZ	SimpleComm	1097	11646	549	11246	548	400

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

since benefits from both exist. TinyThread binds the benefits of both event driven and threaded development models, with several new benefits.

4.4.1 Threaded Comparison

TinyThread has a distinct advantage over many purely threaded platforms. The first major benefit is that interrupt handler can be lightweight because of tasks. Second, developers know exactly how much stack space a thread will require. The final benefit is the ability to further optimize or pipeline any threaded algorithm with event driven programming. The final benefit, although not directly caused by TinyThread is the ability to statically analyze the applications for concurrency problems. This final benefit is caused by the language used for development, namely nesC.

First, developers need not use threads for split interrupt handlers into top and bottom levels. This is a common technique used to reduce the stack overhead by the system. It splits the interrupt handler to a top-half, which executes in the interrupt handler itself, and a bottom half, which does a majority of the processing in a task or thread. Many times, the overall latency of an interrupt is important so the overhead of having to wake up a thread, switch context and execute the bottom

half of an interrupt handler is very inefficient. This occurs in most multithreaded operating system. To meet timing requirements, these OSes are not always able to split their interrupt handler. Without the ability to split the interrupt handler, the stack overhead on every thread increases (this was talked about briefly in Section 3.1. Since most interrupts in TinyOS are cleanly implemented in a top level/bottom level fashion using tasks, TinyOS can create a significantly faster total interrupt latency while maintaining small stack overheads.

The obvious advantage of TinyThread is the distinct usage of *stacksize* to automatically allocate the stack. Not only does this guarantees there will be no wasted RAM, but it also guarantees there will be no stack overflows. Validating this gives developers a true view of their resource consumption. In most research papers on sensornets, the stack usage is never actually measured. This leads to a hidden benefit: when problems bugs occur during testing, it is assured that they are not caused by stack overflows. This also ensures to developers that no matter what state the processor is in, and not matter how much noise there is on interrupt lines, the stack will not overflow.

Another benefit of TinyThread is that threaded applications can be optimized using events. Many times, threading allows developers to rapidly create a procedural application and test it. After rigorous testing, a developer may need to pipeline, or otherwise further optimize an algorithm. Event driven programming would allow a developer to squeeze their application to most efficiently use the resources available. In context of sensornets, if an application was developed using TinyThread to run on an micaZ (with 128kB of ROM and 4kB or RAM) and completely uses the resources. It may be possible to optimized the resources for production on a large scale to fit in a processor with 100KB of ROM and 2kB of RAM; possibly cutting the price in half.

4.4.2 Event Driven Comparison

TinyThread provides several advantages over event driven development models. The first benefit is the rapid time of development. The other clear benefit is the learning curve to develop applications on sensornets. These advantages extend beyond simply introductions to sensornets, to researching new algorithms.

TinyThread's rapid time of application development is actually taken to a significant degree beyond what can be easily measured. The development time for the example applications used during the testing of TinyThread took roughly four-times less time than their event driven equivalents. This is done by a developer experienced writing event driven applications. This really stems from the straight-forward nature of expressing most algorithms.

TinyThread also brings other benefits which are easily overlooked. TinyThread hides the complexities of event driven programming from the developers. For instance, using purely event driven developments, a routine is called to send a message, if the sending queue is full (it has a depth of one), then it returns failure. This failure may be a very real occurence and must be handled by developers to avoid any transient bugs. To combat this type of failure, TinyThread will yield to the system context and tries again later.

4.4.3 Limitations

TinyThread is not the solution for every problem. There are many algorithms, especially in the domain of lower-level networking interfaces, where threading just isn't reasonable. The idea is not to try to build an operating system out of threads, but to build on top of an incredibly efficient system and add the capabilities of threads. There are limitations specific to stack analysis explained in Section 3.2.

As shown in Section 4.3.1, TinyThread does not come without a cost. While

this cost is mainly in RAM, the ROM costs aren't always zero. These costs must be considered whether they can be acceptable for a specific application prior to utilizing TinyThreadİn borderline cases, at least developers can see what the costs are, then optimize later as the resource requirements rise. For instance, a baseline implementation of an application may use three or four threads, but begins to run out of resources. The developer could possibly re-write one or more of the threads into event driven state machines, possibly using less resources, but this still does not negate the resource consumption.

CHAPTER V

RELATED WORK

Much of the work related to TinyThread is deeply involved with the development of embedded Operating Systems (OS). These operating systems run with little or no resource usage and are generally modular. These operating systems can be measured on a variety of different metrics, such as resource consumption, stack consumption, interrupt latency and portability.

5.1 Embedded OS Design

There are two main design philosophies in terms of Embedded Operating Systems: Event-driven and Multi-Threaded. These philosophies are generally followed whether an OS is used or not. If no operating system is used, and only a main loop and interrupts exist, then philosophically, this is an event- (or interrupt-) driven system.

5.1.1 Event-Driven Systems

Event-driven systems are the simplest form of an embedded OS. Although they are not always considered an OS, event-driven embedded systems are sometimes called embedded frameworks because of their simplicity. Event driven programming essentially means that algorithms have to be written in a fashion different than on a PC. Instead of writing procedural code, developers must break the algorithm into several different reactions in a state machine. This is know as stack ripping [1].

Stack ripping can be explained by example. Take for instance, an interrupt driven serial port transmission. In this case, a developer cannot simply write a loop to send 12 bytes because the system must remain responsive. The developer must send the first byte in the main loop, then as the `tx_complete` interrupts occur send the following 11 bytes one at a time. To accomplish this, developers need to use at least one global variable for the index of the data to send next, shown in Figure 23. This example can easily become much more complicated, if the 12 bytes to be sent are actually generated on the fly, possibly from an binary to BCD (binary coded decimal) converter. In this case, the user has to generate the message array in global memory in the main loop prior to starting the cascading transmissions. This example may not seem too complicated, but expand that view to implementing a MAC layer or even Directed Diffusion [15].

Part of the problem with stack ripping is the number of global variables and namespace congestion. If a developer uses the variable name `interruptCounter` in one interrupt handler, another interrupt handler might accidentally use the same variable. If the interrupts can nest, then the application may fault, or in the very least do the wrong thing. This global view can make it very difficult to write modular programs. Take for instance the use of `index` in Figure 23. In the context of the example it appeared clear, but if another routine accidentally uses it, it could cause

```
1  char buffer[] = "Hello World\n";
2  uint8_t index = 0;
3
4  void main(){
5      //Initialize
6      UDR = buffer[index++];
7
8      //Responsive Main Loop
9  }
10
11 ISR(tx_complete){
12     if(index < 12)
13         UDR = buffer[index++];
14 }
```

Figure 23: Interrupt based Serial Transmission.

```
1  char buffer[] = "Hello World\n";
2  uint8_t index = 0;
3
4  void main(){
5      //Initialize
6      create_thread(testThread, stack, sizeof(stack));
7
8      //Responsive Main Loop
9  }
10
11 void testThread(){
12     //Blocking I/O write
13     serialWrite(buffer, 12);
14 }
```

Figure 24: Blocking I/O based Serial Transmission.

several problems.

5.1.2 Multi-Threaded Systems

Multi-Threaded embedded operating systems provide several facilities which event driven operating systems lack. First and foremost is true threading. With threading comes blocking I/O. Blocking I/O is what allows developers to write procedural code in the face of I/O bound routines. Blocking I/O would allow a developer to write a loop to send 10 bytes across a serial link, without blocking the processor. Blocking I/O can allow a system to remain responsive while simplifying the code. This is shown in Figure 24 implementing a possibly equivalent application to the event driven Figure 23.

Blocking I/O and multi-threading does not come without a cost. Each individual thread actually runs in its own context in terms of the processor. This means that each thread must maintain its own stack and its own copy of the processors registers. Outside of the precious RAM that each threads stack consumes, there is a cost of switching between threads. This cost is not too extreme, usually on the order of 100 or so instructions (depending on the processor). After all that cost in resources, preemptive multi-threads can appear like each one has its own CPU. Technically, the threads are actually time-sharing the CPU. This automatic time-sharing causes concurrency problems. Some of these problems occur because a data structure is not in a safe state when another thread attempts to read it. The second thread could crash or simple malfunction.

5.1.3 Threaded InterProcess Communications

To solve concurrency problems caused by preemptive multi-threading, multi-threaded embedded operating systems provide several InterProcess Communication (IPC) mechanisms. The most common is a *Mutex*, sometimes called a *binary semaphore*. Mutexes provide mutually exclusive access to a single data or section of code. This could be multiple threads contending for access to a queue, or multiple threads handshaking using mutexes.

Mailboxes or queues exist to simplify threaded IPC. It can help to simplify thread to interrupt communications and interrupt to thread communications. A mailbox is essentially a buffer, with the ability to block a thread. A mailbox has a finite amount of space. If a mailbox is being unloaded inside of a `tx_complete` interrupt from Figure 23, then a developer's thread only has to write as much data as desired to the mailbox. If the mailbox has no more room to buffer, then the sending thread blocks until some messages are dequeued. Mailboxes are popular as

they provide blocking I/O and hide many global variables.

5.2 Multi-threading and Procedural Code

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

Fibers for TinyOS [30] technically do not require allocating a second stack; there can only be one fiber in the program, and it simply grows the system stack. This stack-less threading model cleanly allows users to use blocking I/O calls without the need for a second stack; the limitation is that there can only one *user* fiber. Similar to protothreads, fibers use `setjmp/longjmp` for their implementation, but instead of jumping back to the main loop, a blocking fiber call actually executes the scheduler (in some limited form) at the point of execution. This allows users to use local variables and block inside of functions, making it unique in terms of stackless threading implementation. Many of the blocking I/O routines described earlier in this thesis (Section 2.3.1) could be ported to support a fiber instead of a thread.

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

Maté [18] is a virtual machine environment for TinyOS. Although in general, the design goals of Maté are quite different from those of TinyThread, there are some similarities worth noting. Like TinyThread, Maté also provides a way for treating split-phase operations as though they were straight-line pseudo-blocking operations. The syntax of writing programs in Maté, however, is quite different from regular nesC, whereas TinyThread is a fully-integrated library extension to nesC, and preserves its syntax.

Kairos [12] is a system that supports "network-as-a-whole" programming of sensornets. Again, while the primary design goals of Kairos are sufficiently different from TinyThread, the two do share a common high-level goal: that of making *programming* of sensornets a more easy and elegant activity. Kairos also supports procedural programming, albeit at the network level, while TinyThread enables procedural programming at the node level.

5.3 Stack Analysis

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

Stacktool [25] provides a method of analysis by actually monitoring the values passed explicitly to the general purpose registers and then out to special purpose registers, such as the stack pointer. Stacktool performs this analysis on a compiled binary file for the ATmel AVR platforms. It internally disassembles the machine code before it processes it. Stack analyzers are traditionally tied to one platform or another, although the techniques are usually more general.

HOIST [24] builds a majority of platform specific points required for stack analysis. Hoist accomplishes this by using a processor (or simulator) as a black box. It is currently limited to 8-bit processors. TinyThread follows the approach laid out by Stacktool, by simulating read/writes to registers. Instead of disassembling the instructions internally, TinyThread relies on *objdump* to perform the disassembly.

5.4 Programming Abstractions

In the recent past, there has been a considerable amount of work on developing usable programming abstractions for sensornet development. **Hood** [31] is an abstraction that allows a node in a sensornet to easily access and interact with other nodes in its neighborhood. These abstractions allow developers to rise above the low-level details of messaging protocols and neighbor lists and design algorithms at a

more abstract level. The primary idea is to create new *programming primitives* that all applications can use without having to deal with the actual implementations. Similarly, the **abstract regions** [30] programming interface provides access to high-level abstractions such as *N-radio hop*, *k-best neighbor*, etc.

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

TinyRPC [21] is a remote procedure call interface for TinyOS. Using TinyRPC, a component can bind to some interface that is actually implemented in a remote node in the 1-hop neighborhood. TinyRPC supports both named bindings and discovered bindings between nodes. Once the binding has been set up, it is easy for nodes to communicate without having to think about messages, node ids, etc. TinyThread provides an exciting new use for TinyRPC. Since nodes can issue remote calls, that means that two threads running on two separate nodes can now use our barrier synchronization abstract to communicate and synchronize directly.

5.5 Types of threading

The term thread is used to describe many different execution models, this section explains the differences between them. A task in TinyOS (considered by many to be an extremely lightweight thread) is not a true thread, since it always executes to completion. TinyOS tasks can be preempted by interrupts, called 'async events'. A task can also be preempted by callbacks, known as signals, which are explicitly called by said task. This simple system of tasks running to executing and a systems of callbacks is a shining example of event based programming. This event based development has proven to be an excellent model for developing sensor network

applications. Tasks provide no way for one task to preempt another, which essentially means it is not a context of execution, and therefor not a thread in the traditional sense.

Threads on modern operating systems utilize a stack to store the current context of the processor when a thread is paused. It is possible to implement extremely light-weight thread-like structures. This stackless operation is implemented using the functions setjmp/longjmp. These routines are the base for *user based threads*, but as Engelschall points out, this only solves the easy half of the thread problem [10]. The harder problem in implementing threads is storing the processor state. In an embedded system without an MMU, this is simply storing the stack and the registers.

If the stack of the thread is not stored, then there are several constraints put on the developer. First, a developer which uses any local variables, these can be corrupted or overwritten during a *blocking* call. Some of this can be overcome using global variables directly, or at least via the *static* keyword. The cost of using only global variables is that it allows, and almost encourages developers to make mistakes. A called function may use local variable, unless it gets automatically inlined by a compiler. This limitation on has greater implications which are not obvious. The second major limitation is that a stackless thread cannot block inside of a routine, but to block longjmp must be called inside of the top level of the thread. This removes much of the ability of developers to layer abstractions on top of these primitives. The programmatic difficulties added by utilizing these stackless threads may be offset by the ability to use blocking I/O. This may be an acceptable tradeoff for certain applications, but it competes against writing modular applications. Stackless threads have been implemented inside of Contiki [8] (called *protothreads*) and an implementation for TinyOS exists(called *fibers*) [30].

Threads with stacks are refered to as threads inside of modern operating sys-

tems, which is the convention follow throughout this thesis. There are two main forms of multi-threading: cooperative multi-threading and preemptive multi-threading. Cooperative Multi-threading is the simpler of the two. Threads still operate with their own context, but the only way for a thread to stop executing is by explicitly yielding its execution via an explicit call to yield or indirectly through a yielding call via blocking I/O routines. Protothreads and TinyOS Fibers are examples of Cooperative Threading. Some cooperative multi-threading implementations only allow yielding to a specified thread, while others allow developers to simply yield to a scheduler. In an embedded system, cooperative threads run in real-time. This real-time execution is a double edge sword. It allows developers a clear view of exactly what variables need to be protected, but on the other hand, it forces developers to be aware of the amount of time any calculations might take, to avoid blocking the processor and dropping packets.

Forcing developers to be fully aware of the length of time a set of library routines takes is rather cumbersome. Preemptive threading solves this problem by forcibly preempting a running thread after some finite period of time. This arbitrary preemption can create concurrency problems, since a thread can then be preempted effectively at any given time. Preemptive reading has the unique ability to overcome a deadlock. For instance, if a task or cooperative thread goes into an endless loop(, the system is deadlocked. Preemptive threads do not deadlock a system when they themselves go into an endless loop. A breakdown of the different features supported in all the previously mentioned threading models can be found in Table II.

The cost of storing the stack is usually guessed, causing explicit problems. Even in commercial embedded operating systems, these stack costs are generally guessed. There are some commercially available tools for calculating stack usage, function by function, but never has it been directly linked into a threading operating system.

Execution Model	Blocking I/O	Real-Time Execution	Local Variables	Multiple Threads
Tasks		√	√	
ProtoThreads	√	√		√
TinyOS Fiber	√	√	√	
Cooperative Multi-threading	√	√	√	√
Preemptive Multi-threading	√		√	√

Table II: Feature lists of different Threading Models.

CHAPTER VI

CONCLUSION

The main goal of TinyThread is to build a new programming paradigm for sensornets while leveraging the de facto standard programming paradigm. The idea is to allow developers to reason and write code in the same fashion they would write on a PC. The goal is to simplify development so much, that programmers with little embedded experience can jump into developing and maintaining applications for sensornets. Multi-threading provides many well founded methods for reasoning, such as CSP [6]. Not to mention, all universities teach threading as part of their undergraduate computer science curriculum. Leveraging these developers directly, with little or no training, can push sensornet applications to new extremes.

TinyThread succeeds in simplifying programming sensornets through multi-threading in a safe fashion. By statically allocating the stacks of each thread, thread overflows are completely prevented. Concurrency problems between threads are completely avoided, since these are cooperative threads (without preemption). Concurrency problems between threads and interrupts are avoided through nesC's concurrency detection. These fix all possible concurrency faults and all stack overflow faults,

which are the biggest problems with threaded development.

While there has been no case study on the matter, there is still some anecdotal evidence that TinyThread radically simplifies development. TinyThread was used to write over 20 different applications. Developing and debugging the equivalent event driven applications would take an incredible amount of time. This in conjunction with the fact that other developers are actively using TinyThread in other research projects reinforces the idea of simplifying development. In many different cases, medium sized TinyThread applications are under 50 lines of code, making them very easy read and understand. Since applications require less code, developers are less likely to make mistakes, and finding mistakes is much easier. Development time, while hard to metric, is reduced by a rough factor of four.

6.1 Future Research

TinyThread pushes the future work farther open than can be speculated. Algorithms which had been thought to be out of practical reach on sensornets, can now be practically researched. The algorithms are only part of the possible future work capable with TinyThread Applications of sensornets can be pushed to new bounds. By easing the barrier of entry for new developers, more ideas and more research could be spurred. This isn't to say that TinyThread is beyond evolutions.

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

As shown in Section 4.3.2 and Chapter 3, the overhead due to interrupts is required to be allocated for each of the threads. This forces threads to use significantly

more memory than their event driven counterparts. Another benefit of a seperate interrupt stack is the ability to implement this efficiently. Since an interrupt handler doesn't damage the current context, it doesn't actually need to swap the full context. The only register which must be swapped back and forth is the stack pointer. In an embedded systems, this must be done with much care if interrupts can possibly nest; since there must also be some state notifying which stack is currently in use.

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

6.2 Implications

The implications of TinyThread may go beyond sensornets. There is nothing besides nesC which bonds TinyThread to sensornets and nesC is not explicitly bound to sensornets. In many practical situations outside of sensornets, TinyThread could possibly have wide acceptance, if it were not for the rapid evolution of nesC. nesC changes every few years, making it wonderful for research, but terrible for production development. At some point in the near future, nesC may lock in with its current syntax so that future versions will be backwards compatible. If this occurs, then TinyThread's influence could spread beyond sensornets.

TinyThread provides a tangible benefit to developers which no other environment provides: the ability to implement algorithms in whichever paradigm is natural to the algorithm. This stems from the ability to mix events and threads. The impli-

cations of this transcends either programming paradigm, since developers can switch between the two at their leisure. This essentially means that algorithms that are naturally reactive will probably be implemented using events, while algorithms that are procedural will probably be implemented with threads. Since algorithms aren't attempted to be forced into a form they don't naturally fit, development of either type will be much easier.

.

BIBLIOGRAPHY

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *USENIX Annual Technical Conference, General Track*, pages 289–302, 2002.

[2] A. Arora, R. Ramnath, E. Ertin, P. Sinha, S. Bapat, V. Naik, V. Kulathumani, H. Zhang, H. Cao, M. Sridharan, S. Kumar, N. Seddon, C. Anderson, T. Herman, N. Trivedi, C. Zhang, M. Nesterenko, R. Shah, S. Kulkarni, M. Aramugam, L. Wang, M. Gouda, Y. ri Choi, D. Culler, P. Dutta, C. Sharp, G. Tolle, M. Grimmer, B. Ferriera, and K. Parker. Exscal: Elements of an extreme scale wireless sensor network. *Real-Time Computing Systems and Applications*, 0:102–108, 2005.

[3] 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. *Mobile Networks and Applications*, 10(4):563–579, Aug. 2005.

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

[5] Chipcon, Texas Instruments. Cc2420 data sheet. http://www.chipcon.com/files/CC2420_Data_Sheet_1_3.pdf, 2005.

[6] E. W. Dijkstra. Notes on structured programming. In O. Dahl, E. Dijkstra, and C. Hoare, editors, *Structured Programming*, number 8 in A.P.I.C. Studies in Data Processing, chapter 1, pages 1–82. Academic Press, 1971.

[7] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN'04)*, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.

[8] A. Dunkels, O. Schmidt, and T. Voigt. Using Protothreads for Sensor Node Programming. In *Proceedings of the REALWSN'05 Workshop on Real-World Wireless Sensor Networks*, Stockholm, Sweden, June 2005.

[9] J. Eidson and L. Kang. IEEE 1588 standard for a precision clock synchronization protocol for networked measurement and control systems. pages 98–105, November 2003.

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

[11] 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 *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2003. ACM Press.

[12] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. In *Proceedings of the International Conference on Distributed Computing in Sensor Systems (DCOSS)*, June 2005.

[13] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *ASPLOS-IX: Proceedings of the ninth international conference on architectural support for programming languages and operating systems*, pages 93–104, New York, NY, USA, 2000. ACM Press.

[14] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd international conference on embedded networked sensor systems*, pages 81–94. ACM Press, 2004.

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

[16] V. A. Kottapalli, A. S. Kiremidjian, J. P. Lynch, E. Carryer, T. W. Kenny, K. H. Law, and Y. Lei. Two-tiered wireless sensor network architecture for structural health monitoring. volume 5057, pages 8–19. SPIE, 2003.

[17] A. Krishnamurthy, D. E. Culler, A. Dusseau, S. C. Goldstein, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in split-c. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 262–273, New York, NY, USA, 1993. ACM Press.

[18] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on architectural support for programming languages and operating systems*, pages 85–95, New York, NY, USA, 2002. ACM Press.

[19] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2004)*, 2004.

[20] K. Lorincz, D. J. Malan, T. R. F. Fulford-Jones, A. Nawoj, A. Clavel, V. Shnayder, G. Mainland, M. Welsh, and S. Moulton. Sensor networks for emergency

response: Challenges and opportunities. *IEEE Pervasive Computing*, 3(4):16–23, 2004.

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

[22] D. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions Communications*, 39(10):1482–1493, October 1991.

[23] Moteiv Corporation. Tmote Sky data sheet. http://moteiv.com/products/docs/tmote-sky-datasheet.pdf, 2005.

[24] J. Regehr and A. Reid. Hoist: a system for automatically deriving static analyzers for embedded systems. In *ASPLOS-XI: Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, pages 133–143, New York, NY, USA, 2004. ACM Press.

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

[26] RFC791. Internet protocol. September 1981. DARPA Internet Program Protocol Specification.

[27] RFC793. Transmission control protocol. September 1981. DARPA Internet Program Protocol Specification.

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

[29] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: scalable threads for internet services. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 268–281, New York, NY, USA, 2003. ACM Press.

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

[31] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 99–110, New York, NY, USA, 2004. ACM Press.

APPENDIX

APPENDIX A

Resource Consumption per Platform

The following tables include all the compilation information about TinyThread. This includes the stack usage from the *main* function (or beginning), the RAM (or data) usage, and the ROM (or program space) usage. The final row in each table includes a measurement of an event driven application. These applications which appear to function the same, are not nearly as flexible as their TinyThread counterparts.

The Tmote results can be found in Tables III, IV, V and VI. The mica2 results can be found in Tables VII, VIII, IX and X. The micaz results can be found in Tables XI, XII, XIII and XIV. At the bottom of each table an event driven version of the same application was analyzed for comparison with the TinyThread version.

Number of Threads	ThreadSafe I/O	RAM(stack)	RAM (data)	ROM (program)
4	√	70	166	3482
3	√	70	148	3456
2	√	70	130	3398
1	√	72	116	3272
4		70	166	3436
3		70	148	3410
2		70	130	3352
1		72	116	3226
Event Driven		96	40	2610

Table III: Resource Consumption of Blink for T-Mote.

Number of Threads	ThreadSafe I/O	RAM(stack)	RAM (data)	ROM (program)
4	√	136	575	13232
3	√	136	557	13206
2	√	136	539	13162
1	√	136	517	12828
4		136	575	13150
3		136	557	13124
2		136	539	13080
1		136	515	12778
Event Driven		142	369	11780

Table IV: Resource Consumption of Bounce for T-Mote.

Number of Threads	ThreadSafe I/O	RAM(stack)	RAM (data)	ROM (program)
4	√	142	2733	13274
3	√	142	2715	13248
2	√	142	2697	13204
1	√	142	2677	12902
4		142	2733	13200
3		142	2715	13174
2		142	2697	13130
1		142	2673	12874
Event Driven		118	2431	11840

Table V: Resource Consumption of Filter for T-Mote.

Number of Threads	ThreadSafe I/O	RAM(stack)	RAM (data)	ROM (program)
4	√	136	641	13186
3	√	136	643	13160
2	√	136	625	13116
1	√	136	605	12790
4		136	661	13104
3		136	643	13078
2		136	625	13034
1		136	601	12716
Event Driven		118	360	12802

Table VI: Resource Consumption of SimpleComm for T-Mote.

Number of Threads	ThreadSafe I/O	RAM(stack)	RAM (data)	ROM (program)
4	√	73	180	2966
3	√	73	165	2936
2	√	73	150	2894
1	√	74	136	2592
4		73	180	2908
3		73	165	2878
2		73	150	2838
1		74	136	2534
Event Driven		45	49	1610

Table VII: Resource Consumption of Blink for Mica2.

Number of Threads	ThreadSafe I/O	RAM(stack)	RAM (data)	ROM (program)
4	√	195	738	12926
3	√	194	723	12890
2	√	194	708	12858
1	√	192	688	12494
4		195	738	12822
3		194	723	12786
2		194	708	12754
1		190	686	12430
Event Driven		135	446	10834

Table VIII: Resource Consumption of Bounce for Mica2.

Number of Threads	ThreadSafe I/O	RAM(stack)	RAM (data)	ROM (program)
4	√	199	2988	13168
3	√	198	2971	13132
2	√	198	2956	13100
1	√	199	2939	12718
4		199	2986	13064
3		198	2971	13028
2		198	2956	12996
1		197	2935	12690
Event Driven		135	2507	11088

Table IX: Resource Consumption of Filter for Mica2.

Number of Threads	ThreadSafe I/O	RAM(stack)	RAM (data)	ROM (program)
4	√	192	914	12758
3	√	191	899	12722
2	√	191	884	12688
1	√	191	865	12330
4		192	914	12662
3		191	899	12626
2		191	884	12592
1		191	865	12254
Event Driven		135	410	11326

Table X: Resource Consumption of SimpleComm for Mica2.

Number of Threads	ThreadSafe I/O	RAM(stack)	RAM (data)	ROM (program)
4	√	73	180	2996
3	√	73	165	2966
2	√	73	150	2924
1	√	74	136	2622
4		73	180	2938
3		73	165	2908
2		73	150	2868
1		74	136	2564
Event Driven		45	49	1640

Table XI: Resource Consumption of Blink for MicaZ.

Number of Threads	ThreadSafe I/O	RAM(stack)	RAM (data)	ROM (program)
4	√	230	723	12280
3	√	230	708	12248
2	√	230	693	12218
1	√	221	667	11848
4		230	723	12176
3		230	708	12144
2		230	693	12114
1		219	665	11784
Event Driven		138	390	10010

Table XII: Resource Consumption of Bounce for MicaZ.

Number of Threads	ThreadSafe I/O	RAM(stack)	RAM (data)	ROM (program)
4	√	234	3010	12494
3	√	234	2995	12462
2	√	234	2980	12432
1	√	228	2951	12044
4		234	3010	12390
3		234	2995	12358
2		234	2980	12328
1		226	2947	12016
Event Driven		138	2451	10258

Table XIII: Resource Consumption of Filter for MicaZ.

Number of Threads	ThreadSafe I/O	RAM(stack)	RAM (data)	ROM (program)
4	√	227	938	12152
3	√	227	923	12120
2	√	227	908	12088
1	√	220	877	11722
4		227	938	12056
3		227	923	12024
2		227	908	11992
1		220	877	11646
Event Driven		163	386	11246

Table XIV: Resource Consumption of SimpleComm for MicaZ.