ABSTRACT

PURVIS, ZANE D. Using Software Thread Integration with TinyOS. (Under the direction of Associate Professor Alexander G. Dean).

Wireless sensor network nodes (motes) often spend time busy-waiting while communicating with peripherals such as radios, analog-to-digital converters (ADCs), memory devices, and sensors. The periods of busy-waiting waste time and energy which are both limited resources for many mote applications. This document presents techniques of using software thread integration (STI) in TinyOS applications to reclaim the idle time and use it for useful processing. The TinyOS scheduler is modified to support the selection and execution of integrated threads and analyzes the impact of integration on task response time. A microphone array sampling application is used to demonstrate the savings. Integrated tasks in the sample application finish 17.7% faster and the application’s active time is reduced by 6.3%.
Using Software Thread Integration with TinyOS

by

Zane D. Purvis

A thesis submitted to the Graduate Faculty of North Carolina State University in partial fulfillment of the requirements for the degree of Master of Science

Computer Engineering

Raleigh, NC

2007

Approved By:

__________________________  _________________________
Dr. Eric Rotenberg        Dr. Suleyman Sair

__________________________  _________________________
Dr. Alexander G. Dean     Dr. Thomas M. Conte
Chair of Advisory Committee
Biography

Zane Dustin Purvis was born June 30, 1979 and grew up in Bennett, North Carolina. He attended college at N.C. State University (NCSU). While enrolled at NCSU, he was an active officer in the ACM-AITP and IEEE student branches. He was also inducted into the honor societies of Eta Kappa Nu (HKN), Tau Beta Pi (TBP), and Phi Kappa Phi (ΦΚΦ). At his college graduation, he received the honor of “Outstanding Double Major” and obtained bachelor’s of science degrees in Computer Engineering and Electrical Engineering, summa cum laude. During his graduate career at NCSU, Zane became active in PackMUG: NCSU’s Apple User Group and a volunteer assistant for many of NCSU’s scuba courses in the Physical Education Department. Zane has worked for EMJ Embedded Systems, IBM, Intel, and Sony-Ericsson. He has served as a Teaching Assistant for computer architecture and embedded systems courses. In addition to his engineering-related work, he has helped to develop educational materials for the NCSU Academic Aquatics programs. In his spare time, Zane enjoys horseback riding, rock climbing, and scuba diving.
Acknowledgements

Thank you for the help, support, and encouragement of Sandy Bronson, Larry Brown, Paul Bryan, Matt Craver, Meghan Hegarty, Angel Johnson, Zeb Kaylor, Kyle Luthy, Leonardo Mattos, Dr. Sichitiu, Won So, and Dr. Wallace.
# Contents

List of Figures ........................................ vi
List of Tables ........................................... vii

1 Introduction ........................................... 1

2 Related Work .......................................... 3
   2.1 Mote Software Systems ......................... 3
   2.2 Mote Power Management ......................... 4
   2.3 Software Thread Integration (STI) ............... 4

3 TOSSTI: TinyOS with Software Thread Integration Support .......... 6
   3.1 Description of TinyOS Scheduler ............... 7
   3.2 Description of TOSSTI Schedulers ............... 8
      3.2.1 Dynamic TOSSTI Scheduler ............... 9
      3.2.2 Static TOSSTI Scheduler ................. 10
      3.2.3 Scheduler Response Times ................. 12
   3.3 Declaring and Posting Merged Tasks in nesC .......... 12
      3.3.1 Processing the Application’s C Code and Generating TIDs .......... 14
      3.3.2 Declaring Integrated Tasks ................. 15
      3.3.3 Using the TOSSTI Script to Generate a TOSSTI Application .......... 15

4 Sample TOSSTI Application ................................ 18
   4.1 MicSampler TinyOS Application .................. 18
      4.1.1 Mote Hardware ............................ 18
      4.1.2 Software .................................. 21
   4.2 Finding Tasks .................................. 23
   4.3 Integrating Tasks ................................ 24
      4.3.1 Generating AVR Assembly Code ............. 24
      4.3.2 Constructing Control Dependence Graphs (CDGs) .......... 25
      4.3.3 Performing Software Thread Integration .......... 26
      4.3.4 Analysis .................................. 28

5 Conclusion ........................................... 32
List of Figures

1.1 Sample mote application timeline without and with STI ............................................. 1

3.1 TOSSTI Toolchain .......................................................................................................... 7
3.2 Diagram of TinyOS scheduler .......................................................................................... 8
3.3 Diagram of Dynamic TOSSTI scheduler ......................................................................... 9
3.4 Diagram of Static TOSSTI Scheduler ............................................................................. 11

4.1 Photo of homemade Mica2 mote .................................................................................. 19
4.2 Photo of homemade MicaZ mote .................................................................................. 20
4.3 Wiring the CC2420EM module to a breadboard ............................................................ 21
4.4 MicSampler software components ............................................................................... 21
4.5 CDG for getSamplesTask ............................................................................................. 25
4.6 CDG for startSend ........................................................................................................ 26
4.7 CFG for startSend, showing cross-jumping .................................................................. 27
4.8 CDG of Integrated getSamplesTask and startSend ..................................................... 28
4.9 Oscilloscope screenshots of task execution times .......................................................... 29
4.10 Oscilloscope screenshots of active time using dynamic scheduler .............................. 29
4.11 Oscilloscope screenshot of active time using static scheduler .................................... 30
List of Tables

4.1 Wiring interface of ATmega128 to CC2420 ................. 20
4.2 Program memory usage for versions of MicSampler application .................. 31
Chapter 1

Introduction

The nodes of common wireless sensor networks (WSNs) motes often have frequent periods of busy-waiting: communicating with radios, communicating with sensors, communicating with flash memory, etc. The wait times in these situations is so small that the cost of a context switch to another task is prohibitive, but useful work could be performed during those times using Software Thread Integration (STI). STI is a compiler technique for producing fine-grained concurrency on processors without additional hardware for fast context switches [8]. The motivation behind the work discussed in this document is to provide a software system to ease the use of STI with a common wireless sensor network (WSN) operating system. Figure 1.1 illustrates how STI can be used to reclaim the idle time that is common when communicating with the mote radio to more efficiently use the mote’s active time. In the figure, you can see how a sample mote application spends time
doing sensing, processing data, and transmitting data followed by a relatively long idle time where the mote goes to a low-power mode. Inside the Transmit task there are many small windows of idle time where the processor is simply waiting for a response back from the radio. After integrating the Transmit and Processing tasks using STI, the idle time that had been present in the Transmit task is replaced with useful work from the Processing task. Because in the STI-version of the application Processing and Transmit tasks are interleaved with each other, the two tasks are completed earlier than if they had been executed serially, as in the original application. Since all tasks are completed sooner, the processor can go to a low power mode sooner, thus improving energy conservation. Idle time spent communicating with other SPI-connected peripherals and waiting for ADC results can also be reclaimed. By more efficiently using the mote’s active time, the mote’s idle time can be maximized resulting in increased battery life.

Chapter 2 discusses other research on software systems for wireless sensor networks, power and energy management for wireless sensor networks, and STI. Chapter 3 describes a software system for easily using STI with TinyOS applications. Chapter 4 describes an application built using TinyOS and STI. Chapter 5 discusses what was learned from this work, and future work.
Chapter 2

Related Work

Wireless sensor networks are currently a very active area of research for computer engineers, computer scientists, electrical engineers, and network engineers. Major areas of research and development include hardware systems, software systems, and power management.

2.1 Mote Software Systems

Several software libraries or so-called operating systems are available for wireless sensor networks.

One popular and promising mote operating system is MANTIS (MultimodAl NeTworks of In-situ Sensors). MANTIS is a project of the University of Colorado at Boulder. It provides a Unix-like concurrency module [23] and is coded in standard C code [20].

The Contiki operating system is, like MANTIS, coded in standard C, but uses protothreads for much of its concurrency [9]. Protothreads are lightweight stackless threads that ease the writing of blocking event-handlers by eliminating/hiding complex state machines [10][11]. Contiki uses an IP networking stack [21].

TinyOS is the current platform of choice for sensor network research [20]. TinyOS is programmed using nesC (Networked Embedded System C) and uses a simple run-to-completion, first-in-first-out concurrency model for any computations that may be time-consuming [14]. Tasks may be preempted by a hardware event, however, as expected. TinyOS’s designers try to make hardware differences between mote platforms invisible to the programmer. Several methods are employed by TinyOS to mask hardware differences from the programmer and provide platform independence for applications:

- component-oriented programming is facilitated by use of the nesC language
• hardware-specific files are separated from platform-generic files

• generic names are given to hardware-specific registers and variables

• a stateless hardware presentation layer (HPL) provides abstracted hardware access [15] [12]

  [20] discusses the interaction of network layers in TinyOS, the various networking and abstraction layers, and how power management is used by various network stack implementations.

  For an excellent description of current operating systems available for wireless sensor networks, see [21].

2.2 Mote Power Management

Power and energy management for motes is an active area of research because motes are deployed in remote locations for long periods of time without changes of batteries. In order to maximize the lifetime of a wireless sensor network (WSN), power management, energy efficiency, and power source are considerations that anyone deploying a WSN must consider.

Researchers and developers of WSNs seek to save power by employing various techniques. [16] analyzes the power management facilities provided by various small microprocessors, such as those used in motes. Various non-battery energy sources for WSN energy scavenging are suggested in [26].

Using energy-mindful MAC and routing algorithms are also common themes in WSN research. [20] compares power management techniques used by various network stack implementations available for TinyOS. [4] describes the SEESAW MAC protocol that attempts to maximize the lifetime of a WSN by using an asynchronous-asymmetric MAC protocol where nodes may take on one of three roles in the network, depending on traffic patterns. [5] describes X-MAC, a very clever MAC suitable for use with packetized radios like the Chipcon CC2420 which uses preamble packets and acknowledgements to shorten the active listening time of the motes and makes up for many of the shortcomings of B-MAC which relies on radios with lower power listening mode [25].

2.3 Software Thread Integration (STI)

Software Thread Integration (STI) is a compiler technique for producing fine-grained concurrency on processors without additional hardware for fast context switches at the expense of larger code size. STI is able to utilize regions of processor idle time that cannot be utilized via context
switches [7]. STI has been used for hardware-to-software migration (HSM), where relatively small sections of idle waiting are common [8]. Chapter 2 of [8] describes in detail the process of performing STI using program dependence graphs (PDGs). [6] describes an HSM application of STI for generating video signals.

In [13], STI is used in the kernel of AvrX to perform cryptographic operations (RC4 and RC5) during TDMA communication, using the idle time between scheduled communication events to perform computations.

Venugopalan uses STI to improve energy efficiency of motes in [30]. However, his system uses MCU to MCU communication via SPI, rather than using actual radios. He instead used the radio’s data sheet to calculate power consumption characteristics, though his MCU-to-MCU system did not completely simulate communication with a radio — only data transmission. His system also does not use a mote operating system/library package.
Chapter 3

TOSSTI: TinyOS with Software Thread Integration Support

*TOSSTI* is a tool set for easily adding facilities for Software Thread Integration (STI) to TinyOS. It includes methods for:

- replacing the default TinyOS 1.1.x scheduler
- declaring nesC tasks that may be integrated with other nesC tasks
- processing a TinyOS application’s code to add STI functionality

When using TOSSTI, the ability to run the non-integrated version of the application and be debugged using the standard TinyOS tools is maintained.

Figure 3.1 gives a graphical overview of the tools used when building an application using TOSSTI:

- the TinyOS application is developed using nesC and compiled using the standard `make` tools included with the TinyOS distribution
- the application is executed and analyzed by the programmer using a debugger and task analysis tools to determine what tasks are good candidates for STI
- the programmer adds TOSSTI mark-ups to the original TinyOS application, and rebuilds
- tasks are integrated using thrint, integrated tasks are declared in an ITF file, and the application is processed by the TOSSTI script include the TOSSTI scheduler
- `gcc` is used to compile the STI code and TOSSTI version of the application into a single executable
This chapter will describe the default TinyOS scheduler, the two TOSSTI schedulers, how to STI tasks are declared in nesC, and processing a TinyOS application’s code to generate an STI version of the application.

### 3.1 Description of TinyOS Scheduler

The scheduler used by TinyOS 1.1.x works as a FIFO queue, running each task to completion. There is no notion of priority among tasks and no task can preempt another task. The only way a running task can be preempted is by an interrupt being serviced. These interrupt service routines are the only form of concurrency available in TinyOS 1.1.x.

In order for TinyOS to support integrated threads, there must be a way to recognize which tasks have been integrated, and the TinyOS task scheduler must be modified to identify when an integrated version of several tasks is available and run it.

The TinyOS 1.1.x scheduler is implemented as a circular FIFO queue and a call to `TOSH_run_next_task()` executes and removes the function at the head of the task queue. The queue is implemented as an array (`TOS_queue`) of C structs with a single field `tp`, which is a pointer to a task. Figure 3.2 shows a conceptual diagram of the TinyOS scheduler. Tasks are C functions that have `void` as both the argument and return type. The array is indexed using
two variables: `TOSH_sched_full` is the array index of the first used position of the queue, and `TOSH_sched_free` is the array index of the first free entry in the queue. The first task to be executed is at `TOS_queue[TOSH_sched_full]`. If `TOS_queue[TOSH_sched_free].tp` is non-null, then the queue is full; otherwise, a new task is added to the queue with the code `TOS_queue[TOSH_sched_free].tp = function_name;`. The size of the queue is defined with the constant `TOSH_MAX_TASKS`, which must be a power of two for efficient modulo arithmetic for wrapping the array indexes beyond the size of the array.

### 3.2 Description of TOSSTI Schedulers

In order to run integrated threads in TinyOS, the scheduler must be modified to determine when an integrated version of multiple tasks is available. Previous schedulers for STI systems utilized two queues because there was an obvious distinction between host/primary and guest/secondary threads: a task would only be posted to the queue to which it belonged, and only the heads of the two queues would be examined when determining whether an integrated version of threads should be executed [6][13][30][31]. This design has an $O(1)$ scheduler. In a multi-queued system, one queue may constantly have fewer tasks in it, resulting in that category of task getting an artificially higher priority. A multiple-queue scheduling system would break TinyOS’s single-priority scheduling semantics because tasks in the consistently shorter queue would get an artificially elevated/reduced priority. This may lead to starvation problems. In TOSSTI, like the TinyOS default scheduler, a single queue is used for all tasks. By using a single queue, the developer does not need to worry about classifying each thread in the system into a category. More importantly, in the single-queued system, starvation problems are eliminated and TinyOS’s nesC task scheduling semantics and syntax do not need to be changed.

Two schedulers have been developed for using STI with TinyOS: a dynamic scheduler
and a static scheduler.

### 3.2.1 Dynamic TOSSTI Scheduler

In the dynamic TOSSTI scheduler, the TinyOS function `TOSH_run_next_task()` has been replaced with `TOSSTI_run_next_task()` and the `TOS_queue` array has been replaced by `TOSSTI_queue`, which is similar to `TOS_queue` except it is an array of `TOSSTI_sched_entry_t` type structures. The definition of `TOSSTI_sched_entry_t` is shown below:

```c
struct TOSSTI_sched_entry_st {
    void (*tp)(void);  // pointer to the task.
    uint8_t tid;      // Task ID. 0 if not TOSSTI
    bool has_run;     // Has this task already been run?
};

typedef struct TOSSTI_sched_entry_st TOSSTI_sched_entry_t;
```

The `tp` field is the same as in the TinyOS scheduler and is simply a pointer to a task/function to be executed.

![Diagram of Dynamic TOSSTI scheduler](image)

**Figure 3.3:** Diagram of Dynamic TOSSTI scheduler

Figure 3.3 shows a conceptual diagram of how the dynamic TOSSTI scheduler works. In the dynamic TOSSTI scheduler, a `tid` field has been added to serve as a unique identifier for each integrated task. When determining what task to run, the dynamic TOSSTI scheduler looks at the head of the queue (`TOSSTI_queue[TOSSTI_sched_full]`). If that entry has a `tid` of 0, then the task is removed from the queue and executed just like in the TinyOS scheduler. However, if the `tid` field is non-zero, it continues searching the queue for another thread with a non-zero `tid`, which may have been integrated with the task. If the scheduler finds another task with a non-zero `tid`, the values of the `tid` fields are bitwise-ORed together and the result is used as an index into the `TOSSTI_integrated_threads` array. If that position of the `TOSSTI_integrated_threads` array is non-null, then an integrated version of the tasks exists at the address pointed to by that entry.
in the TOSSTI_integrated_threads array which should be executed. The scheduler continues searching for another thread until there are no more integrated tasks in the queue.

Because the search for integrated tasks extends beyond the head of the queue, if an integrated version of several tasks is found and executed, then the tasks will no longer be executed in a FIFO order. Rather than remove each of the tasks that have been executed by the integrated thread, a has_run flag is used to indicate to future invocations of TOSSTI_run_next_task() that a task has already been executed out-of-order. Later, when TOSSTI_run_next_task() looks at the head of the queue, and finds that the task has already been run, it simply removes that entry from the queue and returns true, signaling the caller that it actually did execute a task. Using the has_run flag to defer removing executed tasks until later saves the clock cycles that would have otherwise been used to shift other tasks in the queue forward when the executed task is removed.

So that the scheduler does not waste time searching the queue for other integrated tasks when there may not be any, a bit-map called “TOSSTI_locator” is used to indicate the relative position of TOSSTI-tasks in the queue. Bit 0 of TOSSTI_locator corresponds to whether the task at the head of the queue is a TOSSTI-task: a zero indicates that it is a traditional (never integrated) task, a one indicates that it was declared as an integrated task. (More on declaring TOSSTI-tasks in nesC can be found in section 3.3.) Each time a task is removed from the queue, the TOSSTI_locator is right-shifted by one. Whenever a task is added to the queue, the bit corresponding to the position of the new task with respect to the head of the queue is set if the task has a non-zero tid. TOSSTI_locator is used as a boolean to stop the queue search for another integrated task and is used to determine which positions in the queue to search for an integrated task. This scheduler has a worst-case runtime complexity of $O(n)$.

When compared to the standard TinyOS scheduler, the dynamic TOSSTI scheduler uses $2 \times$ more RAM for queue storage on the AVR architecture, but that is actually relatively minimal when considering that instead of two bytes per queue entry as in TinyOS, TOSSTI uses only four bytes, plus one additional byte for the locator bitmap, when using the default queue size of eight. The lookup table that stores the addresses of integrated tasks also requires two bytes per entry, but could be stored in program memory or ROM in systems where tasks do not get added at runtime.

### 3.2.2 Static TOSSTI Scheduler

A second scheduler for using STI with TinyOS has been developed, called the static TOSSTI scheduler. The dynamic TOSSTI scheduler discussed in section 3.2.1 determines whether
and which integrated version of tasks should be executed when the scheduler is fetching a new task from the queue. The static scheduler, however, modifies the queue to include the integrated versions of tasks when a task is posted. The static scheduler’s post function uses a static C array, with one element for each task that has been declared as a TOSSTI task in the system. The array is indexed using a task’s tid and is used to store the position in the TOS_queue array where the previous instance of the posted task was stored. If a new task is being posted to the scheduler, and a task that it has been integrated with has been posted and is still in the queue, the previously posted task’s tp field in the queue is updated to point to the STI version of the tasks, rather than the discrete version of the task, by the post function. Aside from changes in the post operation, all of TinyOS’s other scheduling functions and data structures remain unchanged when using the static TOSSTI scheduler.

Figure 3.4: Scheduler queue before (a) and after (b) posting an integrated task, task-D, to the static TOSSTI scheduler

Figure 3.4 illustrates what occurs when a TOSSTI task is added to the static TOSSTI scheduler. In this particular application, there exists a task named sti-BD which is an integrated version of tasks task-B and task-D. In figure 3.4a, you can see the status of the scheduler queue and the where array after three tasks have been posted: task-A, task-B, and task-C. The entry in the where array corresponding to B points to the location in the queue where task-B was last posted. In figure 3.4b, task-D has just been posted, and since task-D is integrated with task-B, the queue entry for task-B has been replaced with the STI version of the two tasks, sti-BD. The scheduler’s post() function knows that sti-BD is an STI version of task-B and task-D because it was specified as such in the integrated tasks file which is discussed in section 3.3.2.

The static TOSSTI scheduler uses considerably less RAM than the dynamic scheduler. In fact, it requires only one additional byte RAM per TOSSTI task than the default TinyOS scheduler. The code size increase for the static TOSSTI scheduler is also considerably smaller than the dynamic TOSSTI scheduler when few tasks have been integrated: finding an integrated version of two tasks simply involves a C switch and if statements.
3.2.3 Scheduler Response Times

For the original TinyOS FIFO scheduler, if $T_i$ is the execution time of task $i$ and $T_0$ is the time remaining in the current task, the response time for task $n$ is shown in equation 3.1.

$$R(n) = T_0 + \sum_{i=1}^{n} T_i$$ (3.1)

For the TOSSTI schedulers, calculating a response time for a task is not quite as simple because some tasks are removed and replaced as the queue is updated. If $I$ is the set of tasks that have been placed in the queue before task $n$, and have been replaced with an STI-version, and $S$ is the set of integrated (STI) tasks that replaced them, then the response time for task $n$ using a TOSSTI scheduler is as shown in equation 3.2.

$$R(n) = T_0 + \sum_{i=1}^{n} T_i - \sum_{\alpha \in I} T_\alpha + \sum_{\alpha \in S} T_\alpha$$ (3.2)

3.3 Declaring and Posting Merged Tasks in nesC

The nesC programming language used by TinyOS provides the following syntax for defining a task named “myReallySweetTask”:

```nesC
task void myReallySweetTask() {
    // a fun mix of C and nesC code here
}
```

and this syntax for posting the task in the queue to be executed later:

```nesC
post myReallySweetTask();
```

As you can see, neither of these allow any sort of argument passing which would be useful to pass a tid value or otherwise indicate that a task has been merged with other tasks to form an integrated thread [14]. Because the nesC compiler generates C code, a user could hand-modify the generated C code, picking out all posts of tasks they have integrated and replacing them with posts that include tids. Hand-modifying the generated C code is less than ideal and does not do a very good job of integrating with the TinyOS application: whenever the application is recompiled, the hand-modifications will need to be redone. The TOSSTI C preprocessor macro makes it easy to mark integrated tasks.

In order to use the TOSSTI macro, the .nc file containing the module with the integrated task must have the line “#include "TOSSTI.h"” added to it before the definition of the module.
(Note that using the nesC “includes TOSSTI” directive does not work for this use.) The definition of the task in the nesC code then becomes

```nesC
task void TOSSTI(myReallySweetTask) {
    // a fun mix of C and nesC code here
}
```

and all posts of the task in nesC become

```nesC
post TOSSTI(myReallySweetTask);
```

Any use of `myReallySweetTask` must now be wrapped in a call of the `TOSSTI` macro. If a task is declared using `TOSSTI` but is posted without it (or vice versa), the nesC compiler will issue an error that the task has been “implicitly declared” and that “only tasks can be posted” since the task’s name does not match. The `TOSSTI` macro mangles the name of the task so that other tools (see section 3.3.1) can parse the C code generated by the nesC compiler and pick out which tasks have been, or will be, integrated and assign unique `tid`s to them which will be used by the scheduler.

Using the `TOSSTI` macro will not break any existing TinyOS applications and can be used on any task: there does not need to be an integrated version of it available, yet. The process of building and installing a non-integrated version of a TinyOS application using the `TOSSTI` macro is the same as if there were no uses of `TOSSTI` in the code. In fact, it is suggested that an application be implemented, completely debugged, and verified with `TOSSTI` calls in place before doing any STI on the tasks. The only difference one may notice during the process when compared to a traditional TinyOS application is that in a debugger, instead of seeing a task named as “MyModule$mySweetTask,” a `TOSSTI-task` will be named “MyModule$___TOSSTI___mySweetTask___.”

Behind the scenes, nesC’s `post` operations become calls to the `TOS_post` C function, which takes the address of the function being posted as its only argument. When using `TOSSTI`, all calls to `TOS_post` become calls to `TOSSTI_post`, which takes two arguments: the address of the function being added to the queue and a task identification number or `tid`. Each task that has been marked using the `TOSSTI` macro is assigned a unique `tid` when the application is being built. Non-TOSSTI tasks are assigned a `tid` of 0.

An alternative to providing a macro for declaring integrated tasks is to modify the nesC compiler to support new methods of declaring and posting tasks. Using a macro, however, allows anyone to use `TOSSTI` without applying a patch to their compiler and is less likely to break when the nesC compiler is updated.
3.3.1 Processing the Application’s C Code and Generating TIDs

With a normal TinyOS application, a developer types “make micaz install” to compile and load their application onto their mote (replacing “micaz” with the platform they are using, of course). When building an STI version of a TinyOS application, however, after writing and debugging the source code using the standard TinyOS tools, the application’s code must be processed to add the TOSSTI scheduler (discussed in §3.1) and generate tids for the potentially merged tasks.

When a TinyOS application is compiled using the standard TinyOS “make” system, a build/platform/ directory is created by make. That directory contains an app.c file, a binary, and various other files. The nesC language is compiled to C as an intermediary, which is dumped into the app.c file. To automate the process of transforming the app.c TinyOS application to a TOSSTI application with the new scheduler and support for software integrated threads, a script can be used. The TOSSTI script is written in perl and makes a single pass over the app.c file to generate the TOSSTI version of the file.

The TOSSTI script looks for functions declaring TOSSTI-tasks (those tasks that were declared in nesC with the TOSSTI macro). The nesC compiler generates a function declaration for each task with the prototype “void ModuleName$taskName(void);.” Whenever a single line of the app.c file matches this pattern, the TOSSTI script looks for the presence of the string “___TOSSTI___” in the taskName. If “___TOSSTI___” is found, the script puts the function name (including the module name) into a hash table and assigns the task a tid. The hash table is keyed by the function name, and contains the tid as the value. For the dynamic scheduler, the tids are assigned as sequential powers of two (e.g., 1, 2, 4, etc) for the dynamic scheduler.

The TOSSTI script also searches for posts of tasks. All tasks posted in a TOSSTI application must use the TOSSTI scheduler, instead of the default TinyOS scheduler. The nesC statement “post workerTask();” is translated to the C statement “TOSH_post(ModuleName$workerTask)” by the nesC compiler. Whenever the script finds a line with a call to TOSH_post(), it replaces it with a call to TOSSTI_post(), with the correct tid as an argument, using a tid of 0 if the task being posted is a non-TOSSTI task. Because of C’s “declare-before-use” rule, function declarations are found before they are called, so tids have already been assigned by the TOSSTI script before it finds any posts of the tasks.

The TOSSTI script must insert the declarations and definitions of the TOSSTI_post(), TOSSTI_init_sched(), and TOSSTI_run_next_task() functions into its output file. These are all declared and defined in TOSSTI.h and TOSSTI.c which the TOSSTI script #includes just
before the definition of “main” in app.c. Calls to TOSH_init_sched(), and TOSH_run_next_task() are replaced with calls to TOSSTI_init_sched() and TOSSTI_run_next_task(), respectively. The TOSSTI script adds C preprocessor line number directives to the output source code to aid in debugging the application.

3.3.2 Declaring Integrated Tasks

After translating the TinyOS function calls in app.c to the equivalent TOSSTI functions in the output file, the TOSSTI script must define the TOSSTI_integrated_tasks array which is consulted by the dynamic scheduler to find integrated versions of tasks that have been posted to the task queue, or generate a custom TOSSTI_post() function if using the static scheduler.

To indicate to the TOSSTI tools which tasks have been merged to form integrated threads, an integrated threads file is used. Below is an example of an integrated threads file:

```
tasks01 = {MyModule$___TOSSTI___task0, MyModule$___TOSSTI___task1}
tasks03 = {MyModule$___TOSSTI___task0, MyOtherModule$___TOSSTI___task3}
```

The first line indicates that an integrated version of task0 of module MyModule and task1 of module MyModule resides in the function named tasks01. The next line indicates task0 of MyModule and task3 of MyOtherModule are integrated in the function tasks03. Note that a task may be integrated with more than one other task, and that tasks do not need to belong to the same nesC module to be integrated with each other.

Below is the grammar of an integrated tasks file.

```
integratedTasksFile → integratedTaskDeclaration
   → integratedTasksFile integratedTaskDeclaration
integratedTaskDeclaration → IntegratedThreadName = { taskNameList } ;
taskNameList → TaskName , TaskName
   → taskNameList , TaskName
```

3.3.3 Using the TOSSTI Script to Generate a TOSSTI Application

After compiling the TinyOS application with the calls to the TOSSTI macro wrapped around potentially integrated task names, a developer will want to run the TOSSTI script to see which tasks have been marked and build the TOSSTI version of the application. Rather than replacing the sched.c file in the tos/system/ directory, TOSSTI stores the equivalent functions in the TOSSTI.c file and the TOSSTI script inserts them into the TinyOS application after it has been compiled to C by the nesC compiler. This eliminates the need to change files in the user’s TinyOS installation.
The TOSSTI script takes several parameters:

– **application=input path** specifies the path to the nesC generated app.c file for the application being built. This will generally be `build/micaz/app.c`

– **output=output path** specifies the path to where to output the TOSSTI version of the application. For example, `--out=build/micaz/tossti_app.c`.

– **tids=output path** specifies the path to where to output a list of task names and the tids that have been associated with them.

– **integrated-tasks=input path** specifies the path to the integrated tasks file describing which tasks have been merged with other tasks. See section 3.3.2.

– **dollars=replacement text** specifies the string used to replace the $ symbol in identifier names. $ is not valid for identifiers in AVR assembly code, so by default, TOSSTI replaces occurrences of ‘$’ with “___”. Use this option to change the substitution. To leave the $’s in the code, use `--dollars='$'`.

– **verbose** has the script run in verbose mode, displaying information that may be useful for debugging.

The arguments can be abbreviated as long as the abbreviation is unique. For example, instead of typing “`--application=build/micaz/app.c`,” the abbreviation “`--app=build/micaz/app.c`” can be used.

The **application** parameter is always required, the other parameters are optional and are used only when necessary.

Immediately after building a TinyOS application, a developer will probably want to get a list of the TOSSTI tasks. This can be done with the command

```
TOSSTI --application=build/micaz/app.c --tid=tossti_thread_ids.tid
```

A list of tasks read from app.c along with their associated tids will be dumped to a file named “tossti_thread_ids.tid.” The tids are listed only for aid in debugging the integrated application and the identification numbers should not be used by the developer except when debugging. Instead, the task names should be consulted when constructing the integrated threads file.

After constructing an integrated threads file, the developer will want to generate C code for their TOSSTI application. This is done with the command

```
TOSSTI --app=build/micaz/app.c --out=tossti_app.c
```
--int=TOSSTIDemo.itf

Here, the integrated threads input file is named “TOSSTIDemo.itf.” The TOSSTI version of the application will be saved in “tossti_app.c” with the TinyOS scheduler replaced with the TOSSTI scheduler along with the array of integrated tasks. It is then up to the developer to compile and link tossti_app.c with the source files containing the integrated versions of the tasks using the gcc compiler tools included with TinyOS.
Chapter 4

Sample TOSSTI Application

4.1 MicSampler TinyOS Application

A TinyOS application named “MicSampler” has been devised to demonstrate using STI in TinyOS with TOSSTI. MicSampler is roughly based on the microphone array sampling application described by Luthy in [19]. Luthy’s application periodically samples eight microphones attached a mote’s analog to digital converter (ADC), and sends those samples to a PC which calculates the direction of a sound detected within a specific frequency range. This has both military and civilian applications, and could be used as a method for acoustically determining the relative locations of other motes in a network, or for aiding people with disabilities [3]. The differences in Luthy’s mote application and the MicSampler implementation are described below.

The time between reading sets of samples is assumed less important than the time spent between reading each sample from the ADC. That is, each microphone must be sampled as quickly after the previous microphone sampling as possible in order to approximate an instantaneous sampling and be able to determine while direction a sound is emanating from.

4.1.1 Mote Hardware

Luthy used a René mote in his application. The mote used in this project is modeled after a micaZ due to the more modern design and availability of parts and software. Currently, the micaZ is the state-of-the-art mote available for research, though the TI MSP430-based telos or tmotes are also used. The micaZ and the tmotes use the CC2420 radio. The micaZ is better suited for this project than the tmotes because the micaZ is based on on AVR MCU, and existing software tools to aid with software thread integration are available for AVR instruction sets. The AVR’s large register
file also may arguably make register allocation for STI easier than the MSP430’s 16\footnote{28}.

The René mote Luthy used is based on the now-deprecated Atmel 90LS8035 AVR microcontroller, running at 4 MHz and uses the TR1000 radio from RF Monolithics\footnote{19}. Originally, this project used a homemade Mica2 mote, which uses an Atmel ATmega128 microcontroller running at 4 MHz, Atmel STK-500, Atmel STK-501, and Chipcon CC1000 radio because we already had parts to build one, and a schematic was available from Crossbow\cite{17}\cite{22}. A photograph of one of the homemade Mica2 motes can be seen in Figure\textcolor{red}{4.1} Although a Mica2-like mote was used to init-

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{homemade_mote}
\caption{Homemade Mica2 mote, using STK500, STK501, ATmega128, and CC1000EM.}
\end{figure}

ially test out TinyOS and determine how suitable it would be for supporting STI, it was determined that a more modern mote architecture might be a better example of using STI in WSNs.

While the mica2 mote has a schematic publicly and freely available, there was no such file available for the more modern micaZ mote. However, by examining the uses of the \texttt{TOSH\_ASSIGN\_PIN} macros in the TinyOS software directory \texttt{...tinyos/platform/micaz/hardware.h}, the ATmega128 physical interface to the CC2420 radio can be determined to be as shown in Table\textcolor{red}{4.1} A photo showing the homemade mote using the CC2420 radio can be seen in Figure\textcolor{red}{4.2} Because the socket on the underside of the CC2420EM module are not common headers, 16 gauge wire was used to make the evaluation module compatible with a standard breadboard. Figure\textcolor{red}{4.3} shows how this wiring was used.
### Table 4.1: Wiring interface of ATmega128 to CC2420

<table>
<thead>
<tr>
<th>CC2420EM Pin</th>
<th>CC2420 Pin</th>
<th>AVR Pin</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1-3</td>
<td>41</td>
<td>A5</td>
<td>VREG_EN</td>
</tr>
<tr>
<td>P1-5</td>
<td>21</td>
<td>A6</td>
<td>CC_RSTN</td>
</tr>
<tr>
<td>P1-6</td>
<td>30</td>
<td>B7</td>
<td>CC_FIFO</td>
</tr>
<tr>
<td>P1-8</td>
<td>29</td>
<td>E6 (INT6)</td>
<td>CC_FIFOP / CC_FIFOP1</td>
</tr>
<tr>
<td>P1-10</td>
<td>28</td>
<td>D6</td>
<td>RADIO_CCA (clear channel assessment)</td>
</tr>
<tr>
<td>P1-12</td>
<td>27</td>
<td>D4</td>
<td>CC_SFD (start of frame detected)</td>
</tr>
<tr>
<td>P1-14</td>
<td>31</td>
<td>B0</td>
<td>CC_CS</td>
</tr>
<tr>
<td>P1-16</td>
<td>32</td>
<td>B1</td>
<td>SPI_SCK</td>
</tr>
<tr>
<td>P1-18</td>
<td>33</td>
<td>B2</td>
<td>MOSI</td>
</tr>
<tr>
<td>P1-20</td>
<td>34</td>
<td>B3</td>
<td>MISO</td>
</tr>
<tr>
<td>P2-9</td>
<td>VTG</td>
<td></td>
<td>3.3 V</td>
</tr>
<tr>
<td>P2-20</td>
<td>GND</td>
<td></td>
<td>Gnd</td>
</tr>
</tbody>
</table>

### Figure 4.2: Homemade MicaZ Mote, using STK500, STK501, ATmega128, and CC2420EM.
4.1.2 Software

Luthy’s mote was programmed using AVR assembly language. The demonstration application for this project was written in nesC to take advantage of and showcase the use of existing TinyOS modules and tools. Some time-critical parts were written using inline AVR assembly.

Overview of MicSampler Application

![Diagram of MicSampler Application](image)

**Figure 4.4:** Components used in MicSampler as generated using nesdoc

When the mote boots, the radio is configured, the ADC is configured, and the MicroTimer
is configured to signal an interrupt periodically, at desired the sample rate. Whenever that interrupt occurs, the TinyOS signal `MicroTimer.timerFired` is executed which does two things: read all eight channels from the AVR’s ADC and transmit the eight ADC samples from the previous sample period over the mote’s CC2420 radio. The sample and transmit operations are performed in TinyOS tasks. Figure 4.4 shows the component interaction diagram generated for the MicSampler application by the `nesdoc` tool.

**Description of MicSampler Files**

MicSampler consists of four original nesC files: MicSampler.nc, MicSamplerM.nc, MicArray.nc, and MicArrayM.nc. It also uses modified versions of the TinyOS CC2420RadioM.nc, HPLCC2420M.nc, and the HighFrequencySampling demo application’s MicroTimerM.nc. It also consists of the C header file MicArrayMsg.h.

**MicSampler.nc** Configuration describing the components of the application, and wiring between them.

**MicSamplerM.nc** Module implementing the interfaces indicated in MicSampler.nc and implementation of top-level program logic.

**MicArray.nc** Interface describing the basic interface (commands and events) required to be implemented by a MicArray component.

**MicArrayM.nc** Module implementing each of the commands specified in MicArray.nc using the on-chip ADC of the ATmega128 of a micaZ mote.

**CC2420RadioM.nc** Modified from the standard TinyOS system file to include the TOSSTI markups and with no software-based initial back-off when preparing to transmit.

**MicroTimerM.nc** Modified from the demonstration HighFrequencySampling TinyOS application so that the microsecond-resolution timer can be used at the same time as the micaZ’s CC2420 radio, a limitation in the demo application.

**MicArrayMsg.h** Description of the MicArray packet format for transmitting a collection of samples to other motes.
4.2 Finding Tasks

After the basic TinyOS version of MicSampler was designed, coded, and tested, the TOSSTI version was made. The first step in creating the TOSSTI version was determining what tasks were in the system, and which ones to integrate. TinyOS includes a tool for listing tasks in a system, taskCount.pl, which claims, “Until we have ‘ps,’ we might as well have this.” To find a list of tasks in the current TinyOS application directory use the command

```bash
../../tools/scripts/taskCount.pl micaz.
```

It will list all the tasks in the application. For the MicSampler application, the following was displayed:

<table>
<thead>
<tr>
<th>Addr</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x1190</td>
<td>AMStandard.sendTask</td>
</tr>
<tr>
<td>0x285e</td>
<td>CC2420ControlM.PostOscillatorOn</td>
</tr>
<tr>
<td>0x8d4</td>
<td>CC2420ControlM.taskInitDone</td>
</tr>
<tr>
<td>0x222e</td>
<td>CC2420RadioM.PacketRcvd</td>
</tr>
<tr>
<td>0x2206</td>
<td>CC2420RadioM.PacketSent</td>
</tr>
<tr>
<td>0x14e6</td>
<td>CC2420RadioM.startSend</td>
</tr>
<tr>
<td>0x1e90</td>
<td>CC2420RadioM.delayedRXFIFOTask</td>
</tr>
<tr>
<td>0x32d6</td>
<td>FramerAckM.SendAckTask</td>
</tr>
<tr>
<td>0x3210</td>
<td>FramerM.PacketRcvd</td>
</tr>
<tr>
<td>0x133a</td>
<td>FramerM.PacketSent</td>
</tr>
<tr>
<td>0x333c</td>
<td>FramerM.PacketUnknown</td>
</tr>
<tr>
<td>0x1fde</td>
<td>HPLCC2420FIFOM.signalRXdone</td>
</tr>
<tr>
<td>0x163a</td>
<td>HPLCC2420FIFOM.signalTXdone</td>
</tr>
<tr>
<td>0x2a7e</td>
<td>HPLCC2420M.signalRAMWr</td>
</tr>
<tr>
<td>0xda2a</td>
<td>HPLPowerManagementM.doAdjustment</td>
</tr>
<tr>
<td>0x1a24</td>
<td>MicArrayM.getSamplesTask</td>
</tr>
<tr>
<td>0x1b2</td>
<td>MicSamplerM.timerFiredTask</td>
</tr>
<tr>
<td>0x253e</td>
<td>TimerM.HandleFire</td>
</tr>
<tr>
<td>0x26fa</td>
<td>TimerM.signalOneTimer</td>
</tr>
</tbody>
</table>

19 tasks

After listing the tasks in the system, to determine which tasks were ready to run at the same time, one can set a breakpoint at each task using gdb and the Atmel JTAG In-Circuit Emulator (JTAG-ICE). When the breakpoint is reached, use gdb to print the TOSH_queue array and make note of which tasks are consistently present in the queue at the same time. Those tasks are good candidates for integration if they meet the following requirements:

- data-independent with other tasks
Code consisting of busy-wait loops are common in tasks that communicate with the radio (transmit and receive) code that communicates with common sensors attached to the SPI bus, and external Flash memory. Tasks that read from the ADC also commonly have busy wait loops: on the AVR architecture the on-chip successive-approximation ADC can take 14 ADC clock cycles to perform a conversion, depending on how it is configured [1]. Looping constructs beyond the region to be integrated can have an indeterminate number iterations: only loops that overlap with the other thread during integration need to have known iteration counts. The same holds true for function calls beyond the integration region.

Changes to MicSampler Application for TOSSTI

The sample-reading task (getSamplesTask) and the TinyOS CC2420FIFOM.startSend tasks were marked with the TOSSTI macro to indicate that there may be a version of them existing in the system that has been integrated with another task. If, during compilation, the TOSSTI.pl script is used, then a TOSSTI-enabled application can be generated, otherwise, if only the standard TinyOS build tools are used, the application is built without STI enabled.

4.3 Integrating Tasks

4.3.1 Generating AVR Assembly Code

For the MicSampler application, the tasks CC2420RadioM$startSend and MicArrayM$getSamplesTask were ready to run at the same time, so they were good candidates for integration. After picking out which tasks to integrate, assembly code for those functions was necessary. To generate assembly code for the tasks, avr-gcc can be used to compile build/micaz/app.c. However, ncc, the nesC compiler front end generates C code with $'s in identifiers. Since $ is used to indicate hexadecimal numbers in AVR assembly language, those cause problems when compiling. To remove them, either the TOSSTI script can be used, or the command

sed "s/$/__/g" build/micaz/app.c > app_free.c
can be used. The same optimization options used by the TinyOS build system are used:

- Os -finline-limit=10000.

### 4.3.2 Constructing Control Dependence Graphs (CDGs)

![CDG for getSamplesTask](image)

**Figure 4.5:** CDG for `getSamplesTask`

Generating the control dependence graph for `getSamplesTask` was relatively easy: `getSamplesTask` is straight-line code. Its control dependence graph can be seen in Figure 4.5:

- green nodes indicate basic blocks that start an ADC conversion and write an ADC value to memory
- gold blocks are the ADC read operations
- gray blocks are `nop` used to pad away busy-waiting

Generating the control dependence graph for `startSend` was not as easy. First of all, the `startSend` task called several functions which needed to be inlined before the control dependence graph (CDG) could be generated using `thrint` [29]. Modifying the `app_free.c` file so that functions called by `startSend` were inlined by using `gcc`’s `__attribute__((always_inline))` on the function declarations of called functions helps to eliminate the call instruction, and makes integration easier. An alternative to using `gcc` function attributes is converting the function to a C-macro. The `startSend` task called several functions which needed to be inlined before the control dependence graph (CDG) could be generated using `thrint`. The declarations for each of `startSend`’s called functions were modified until `gcc` would inline them, except for the call to `TOS_post`, which came only at the function’s end, and would not interfere with control dependence graph construction.

When using `gcc` to compile a single function to assembly for use as one of the threads to integrate, be sure to specify `-finline-limit=10000` and `-Wall`. The GCC `__attribute__((used))` attribute can be applied to a function to convince GCC’s dead code elimination that the function is, in fact, useful, against its better judgement.
After inlining called functions, \texttt{thrint} is used to construct the CDG for \texttt{startSend} as is seen in figure 4.6:

- blue diamonds represent predicate nodes
- pink ovals represent the entry of a loop, with the conditional in a predicate node below it
- magenta nodes are function calls that were not inlined
- white nodes are simply code blocks

While using \texttt{thrint} to construct the CDG it was noticed that one of the optimizations included with \texttt{avr-gcc}'s \texttt{-Os} optimization setting sometimes generates unstructured code that the version of \texttt{thrint} used for this project cannot understand, although work is being done to add restructuring capabilities to \texttt{thrint}. Luckily, when the \texttt{-fno-crossjumping} option was passed to \texttt{gcc} when compiling \texttt{startSend} to assembly code, \texttt{thrint} was able to accept the code as input. To illustrate how cross-jumping affects the control flow of the code, figure 4.7 shows the control flow graphs of \texttt{startSend} with and without cross-jumping enabled: (a) has a jump into a decision (ID) \cite{24} which does not occur in (b) when cross-jumping is disabled.

### 4.3.3 Performing Software Thread Integration

While \texttt{thrint} was used to form the CDGs, code transformations used for integration were done manually. The \texttt{getSamplesTask} was designated the \texttt{guest} thread since it had internal real-time requirements while the \texttt{startSend} function was designated the \texttt{host} thread.
Figure 4.7: CFG of \texttt{startSend} with (a), and \textit{without} (b) cross-jumping optimization

Because TinyOS/nesc tasks take no arguments, and return void, when doing register allocation, you do not need to worry about calling functions placing arguments in certain registers, and you can use the return registers without concern.

\textbf{Modifications to \texttt{getSamplesTask} for STI}

The function \texttt{getSamplesTask} required no changes for STI that were not done previously for timing purposes: busy-wait loop unrolling, and software pipelining. When the CDG was generated by \texttt{thrint}, however, the start cycle of actual code blocks (as opposed to blocks of \texttt{nop}s) was noted so that it could be determined where in \texttt{startSend}'s CDG to insert each code node.

Also, the register file was partitioned so that register uses would not collide with registers used by the \texttt{startSend} function. Register use was limited as much as possible, without affecting real-time constraints within the function.

\textbf{Modifications to \texttt{startSend} for STI}

The function \texttt{startSend}, in contrast to \texttt{getSamplesTask}, had many busy-wait loops, as well as a loop over the elements of an array (the bytes of the packet to be transmitted over the radio). Before starting integration, the busy-wait loops were unrolled, and padded with \texttt{nop}s. To figure out how many \texttt{nop} instructions were needed, the datasheets for the ATmega128 MCU
and CC2420 radio were consulted as well as experimentally testing with varying numbers of nops until the exact number of clock cycles to complete the SPI writes was known [1][2]. The CDG was, of course, reconstructed after unrolling the busy-wait loops. As each of the code nodes from getSamplesTask was inserted into startSend’s CDG at the correct cycle, getSamplesTask code from nodes that overlapped with the nops from getSamplesTask’s unrolled busy-wait loops could replace those nops. Also, because startSend has a loop over each byte in the array which runs at a frequency other than getSamplesTask’s reads of ADC channels, the loop needed to be peeled five times for the guest code nodes to be inserted at the correct cycles.

Manually using register partitioning for startSend was more complicated than that for getSamplesTask because startSend had to avoid overlapping register uses with those of getSamplesTask. Using the command

```
```

partitioned the registers enough that once the two functions were integrated, there was only one case of a register definition made by getSamplesTask colliding with a register definition made by startSend. Luckily, the value in the register was a constant used by startSend that getSamplesTask overwrote. Another ldi instruction was inserted before the use of that constant so that the integrated function would function correctly.

Figure 4.8 shows the CDG of the software thread integrated version of the two functions.

![Figure 4.8: CDG of Integrated getSamplesTask and startSend](image)

**4.3.4 Analysis**

Figure 4.9 shows oscilloscope screenshots of the time required to execute the tasks getSamplesTask and startSend. To generate the waveforms, the program was modified to turn
on a single output pin on the MCU when the task began execution, and turned off just before that task’s `ret` instruction was executed. You can see from (a) that `getSamplesTask` takes 77.7 $\mu$s to execute and `startSend` takes 200.5 $\mu$s, for a total of 278.2 $\mu$s, including the time required to toggle the output pins. (b) shows that the integrated version of `getSamplesTask` and `startSend` takes only 229.0 $\mu$s to execute. When compared with the discrete versions of the threads, the integrated tasks show a 17.7% reduction in run time.

**Active Time Analysis**

After verifying the speed improvement of the integrated task compared to the discrete tasks, the time the processor was active was measured. To do this, an output pin was turned on when the processor woke up from a sleep mode, which involved modifying each interrupt service routine to turn on the pin when it was executed. The `TOSH_run_task()` function turned off that output pin before executing the AVR `sleep` instruction. Figure 4.10 shows the active time of the
processor (a) without TOSSTI and (b) with TOSSTI while using the dynamic TOSSTI scheduler. The bottom-most waveform, labeled “ACTIVE” shows the active time of the processor, and one can see that its width is (a) 514.0 $\mu$s when not using TOSSTI, and (b) 566.5 $\mu$s when using TOSSTI: that means an *increase* in active time when using STI rather than the expected *reduction*! When looking at how much time is spent actually spent searching for the next task to execute (waveform “SCHED”), (b) which uses STI spends over twice as much time as (a) which is the plain TinyOS implementation of the application. For this application, the runtime overhead of the dynamic TOSSTI scheduler negates the benefits of using STI. In an application with longer tasks, this overhead may be negligible, but since there are many applications where the tasks are short, the runtime overhead needed to be reduced for STI to be feasible for WSN applications. This realization is actually what led to the development of the static TOSSTI scheduler which is described in section [3.2.1].

![Oscilloscope screenshot showing MicSampler application using static TOSSTI scheduler](image)

**Figure 4.11:** Oscilloscope screenshot showing MicSampler application using *static* TOSSTI scheduler

After the static TOSSTI scheduler was developed, the active-time analysis was done again. Figure 4.11 shows the active time of the same application using STI and the static TOSSTI scheduler is 481.5 $\mu$s. When compared to the non-STI version of the application as seen in figure 4.10a, that is a time savings of 32.5 $\mu$s or a decrease of 6.3%. By running the integrated version of the threads rather than the discrete versions using the static TOSSTI scheduler, the processor can go back to a sleep mode 6.3% sooner with this application. With other applications, including those that integrate security features or routing tasks, it is likely that the savings will be even greater [6][13][30].

Even though the runtime overhead using the dynamic TOSSTI scheduler is prohibitive for this application, in systems where new tasks may be introduced to the system at run time, the dynamic scheduler could be useful. While, for the time-being, having new tasks enter the system at runtime may not be feasible, such a situation can occur in larger systems such as those targeted by
So in [27].

**Response Time**

In this application, the `timerFired` task calls the function which posts the `AMStandard.sendTask` then calls the function which posts `getSamplesTask`. `AMStandard.sendTask` calls the function that posts `startSend` task. So, for this application, even though an impressive sequence of function calls is made each time the timer fires, the deepest the task queue generally gets is two tasks deep. The response time for `getSamplesTask` is unchanged here, but the response time for `startSend` gets reduced. According to equation [3.2] the response time for `startSend` in the integrated versions of the application is now

\[
R(\text{startSend}) = (T_{\text{getSamplesTask}} + T_{\text{startSend}}) - (T_{\text{getSamplesTask}} + T_{\text{startSend}}) + T_{\text{sendAndSample}}
\]

**Program Memory Usage**

<table>
<thead>
<tr>
<th></th>
<th>Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-STI App</td>
<td>11,130</td>
</tr>
<tr>
<td>App with Dynamic TOSSTI</td>
<td>13,166</td>
</tr>
<tr>
<td>Static TOSSTI Scheduler</td>
<td>12,948</td>
</tr>
<tr>
<td><code>startSend</code> + CC2420.cmd</td>
<td>306</td>
</tr>
<tr>
<td><code>getSamplesTask</code></td>
<td>570</td>
</tr>
<tr>
<td>sendAndSample function</td>
<td>1,168</td>
</tr>
</tbody>
</table>

STI results in code size expansion because in addition to the original, non integrated versions of the tasks, there are also clones of the task bodies in the integrated task. The modified scheduler also increases code size. However, flash memory is relatively inexpensive, and on-chip program memory for today’s motes is relatively large when considering the limited amount of processing currently performed on the motes themselves. Table 4.2 shows a summary of program memory requirement for the various versions of the MicSampler application. The data was collected using the `avr-objdump` utility. Code sizes for individual scheduler functions is not available because for different versions of the application, the compiler inlines and optimizes away the functions differently.
Chapter 5

Conclusion

This work has shown that using software thread integration with a mote software system such as TinyOS can be done with changes to the TinyOS scheduler, and a method of marking integrated threads, TOSSTI. It also shows an example application that uses software thread integration and TOSSTI to reclaim busy-wait time that is common in many WSN applications. During what was previously busy-wait time, the mote can now perform useful operations, which results in the mote completing its active cycle sooner, meaning it can go back to its idle state and low-power mode sooner, reducing energy consumption. In the sample application, active time was reduced 6.3%, but in other applications, it is likely that the reduction can be even greater.

In the future, TOSSTI can be ported to the new TinyOS 2.0, which provides a more easily accessible method of utilizing a custom scheduler [18]. Additional applications utilizing TOSSTI will demonstrate that using STI on motes can reduce active time even more, resulting in longer battery lifetimes. The thrint application can be modified to perform integration of AVR code and be less error-prone. Automatic register reallocation by thrint would also be a very welcome addition.
Bibliography


