

# TOSSTI: Saving Time and Energy in TinyOS with Software Thread Integration\*

Zane D. Purvis  
zane.purvis@gmail.com  
Center for Efficient, Scalable, and Reliable Computing (CESR)  
North Carolina State University  
Raleigh, NC 27695

Alexander G. Dean  
alex\_dean@ncsu.edu  
Center for Efficient, Scalable, and Reliable Computing (CESR)  
North Carolina State University  
Raleigh, NC 27695

## Abstract

Many wireless sensor nodes (*mot*es) interface with slow peripheral devices, requiring the processor to wait. These delays waste time, energy and power, which are valuable but limited resources on many motes. This paper presents techniques to use software thread integration (STI) in TinyOS applications to recover the idle time for useful processing. We modify the TOS scheduler to support the selection and execution of integrated threads. We analyze the impact of integration on task response time. We demonstrate these methods by applying them to a microphone array sampling application to save computation time and energy. We find that the integrated tasks finish 17.7% faster, reducing application active time (and hence application energy) by 6.3%.

## 1. Introduction

The nodes of common wireless sensor networks (WSNs), called *mot*es, often have frequent periods of busy-waiting: communicating with radios, communicating with sensors, flash memory, A/D converters, etc. The wait times in these situations are so short that the cost of a context switch to another task is prohibitive. 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 [4]. This paper introduces TOSSTI, a software system which uses STI with the common wireless sensor network (WSN) operating system TinyOS.

Figure 1 illustrates how STI can be used to reclaim the idle time that is common when communicating with the mote radio, in order to more efficiently use the mote's active time. A sample mote application spends time 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.

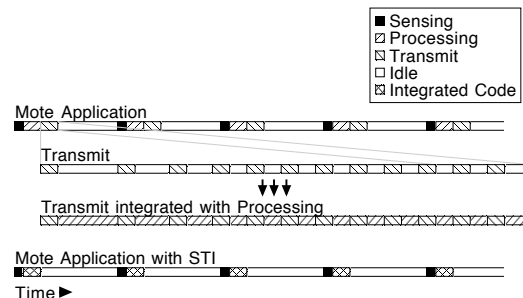


Figure 1. Sample mote application timeline without and with STI

There are multiple benefits from completing the tasks earlier. First, the processor can switch to a low power mode sooner, thus improving energy conservation. By more efficiently using the mote's active time, its idle time rises, boosting battery life. Second, completing the work early benefits real-time systems even if they already meet their deadlines. Earlier task completion improves overall response times, and also allows the system to support more higher-priority processing (e.g. ISRs) without missing deadlines.

An additional benefit of applying STI is that it enhances the concurrency model of the scheduler. TOSSTI enables

\*This work was supported by NSF award CNS-0509162

TinyOS's non-preemptive scheduler to provide the effects of preemption at the task level without incurring the additional context switching penalties resulting from preemptions.

## 2. Related Work

### 2.1. Mote Systems

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. [14] 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 [23]. ICEM [15] is an example of recent work which provides operating system support for power management in Tiny OS. ICEM uses special non-blocking locks (called power locks) to allow applications to expose low-level concurrency.

Using energy-minded MAC and routing algorithms are also common themes in WSN research. [18] compares power management techniques used by various network stack implementations available for TinyOS. [2] 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. [3] 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 [21].

Several software libraries or so-called operating systems are available for wireless sensor networks including, MANTIS from University of Colorado at Boulder [20], Contiki from the Swedish Institute of Computer Science [7], and TinyOS from University of California at Berkeley [10]. For an excellent description of current operating systems available for wireless sensor networks, see [19].

The Contiki operating system is coded in standard C, but uses protothreads for much of its concurrency [7]. Protothreads are lightweight stackless threads that ease the writing of blocking event-handlers by eliminating/hiding complex state machines [8][9]. Protothreads offer *lightweight* context switching between threads (us-

ing C code which compiles to a jump instruction), while STI copies and moves assembly instructions to eliminate the need for most jumps, providing *weightless* context-switching in most places.

TinyOS is the current platform of choice for sensor network research [18]. 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 [12]. 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 [13][10].

### 2.2. Software Thread Integration (STI)

Software Thread Integration (STI) is a compiler technique which provides fine-grained concurrency on processors without additional hardware for fast context switches. STI is able to recover regions of processor idle time that cannot be utilized via context switches [6] due to granularity issues. The fundamental idea behind STI is the *assembly-language-level* compile-time interleaving of multiple functions (from different threads, typically written in C). The resulting implicitly-multithreaded functions lack internal context switching and yield essentially weightless threads. STI duplicates and moves instructions to maintain control-flow and data-flow semantics for correct functionality. STI's weightless context switching can be used to reclaim processor idle time which would be too short to use with traditional context switches (or even interrupt service routines). STI may also provide some performance improvement through the elimination of context switching overhead, but this is a secondary benefit.

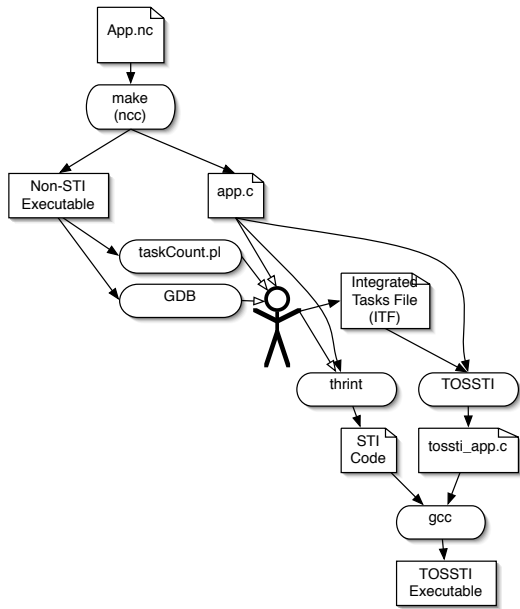
STI has been used for hardware-to-software migration (HSM), where relatively small sections of idle waiting are common [4]. Chapter 2 of [4] describes in detail the process of performing STI using program dependence graphs (PDGs). [5] describes an HSM application of STI for generating video signals. In [11], 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 [24]. 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.

This paper describes a system for incorporating STI into TinyOS applications relatively easily. Using STI with WSN applications can reduce energy consumption and increase

performance.

### 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, and 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. The tools can be found online at <http://www.cesr.ncsu.edu/agdean/tossti>.



**Figure 2. Interaction of tools used when building a TOSSTI application**

Figure 2 gives a graphical overview of the tools used when building an application using TOSSTI. First, the TinyOS application is developed using nesC and compiled using the standard *make* tools included with the TinyOS distribution. Second, the application is executed and analyzed by the programmer using a debugger. Task analysis tools determine which tasks are good candidates for STI. Third, the programmer adds *TOSSTI* mark-ups to the original TinyOS application, and rebuilds it. Fourth, tasks are integrated using *thrint*, integrated tasks are declared in an ITF file, and the application is processed by the *TOSSTI* scheduler to include the *TOSSTI* scheduler. Finally, *gcc* is used

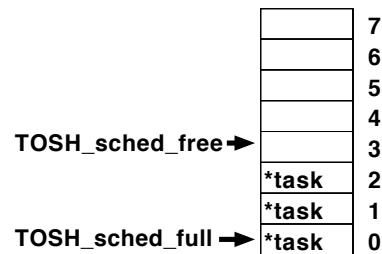
to compile the STI code and *TOSSTI* version of the application into a single executable.

#### 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. TinyOS scheduler queue**

Figure 3 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: *TOS\_sched\_full* is the array index of the first used position of the queue, and *TOS\_sched\_free* is the array index of the first free entry in the queue. The first task to be executed is at *TOS\_queue[TOS\_sched\_full]*. If *TOS\_queue[TOS\_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[TOS\_sched\_free].tp = function;*. 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 ver-

sion 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 [5][11][24][25]. This design has an  $O(1)$  scheduler. 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, which 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, starvation problems are eliminated and TinyOS's nesC task scheduling semantics and syntax do not need to be changed.

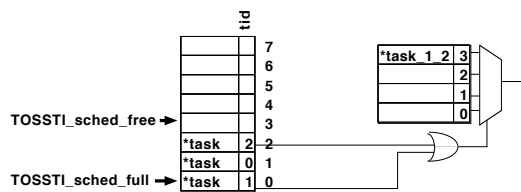
Two schedulers have been developed for using STI with TinyOS: one dynamic and one static.

### 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:

```
typedef struct TOSSTI_sched_entry_st {
    void (*tp)(void);
    uint8_t tid;
    bool has_run;
} 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.



**Figure 4. Dynamic TOSSTI scheduler queue and table**

Figure 4 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.

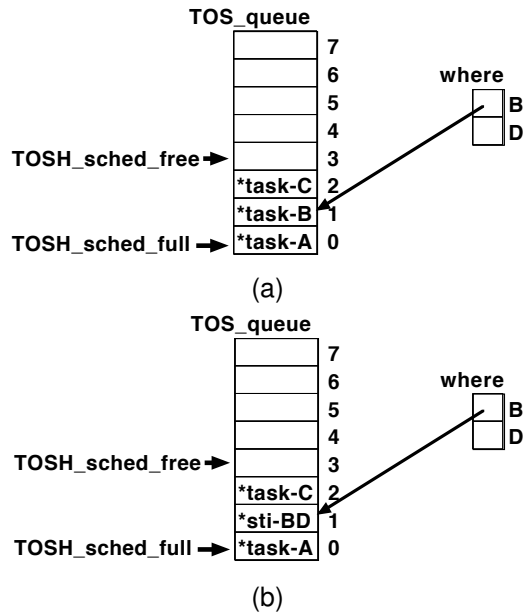
Because the search for integrated tasks extends beyond the head of the queue, if an integrated version of several tasks is found and executed, then the tasks will no longer be executed in a FIFO order. Rather than remove each of the tasks that have been executed by the integrated thread, a *has\_run* 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* indicates if the task at the head of the queue is an integrated TOSSTI-task. 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*. When *TOSSTI\_locator* is 0, no search of the queue for integrated tasks is needed. 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 5. Scheduler queue before (a) and after (b) posting an integrated task, *task-D*, to the static TOSSTI scheduler**

Figure 5 illustrates what occurs when a TOSSTI task is added to the static TOSSTI scheduler. Task *sti-BD* is an integrated version of tasks *task-B* and *task-D*. Figure 5a shows 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 5b, *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 much less RAM than the dynamic scheduler: only one more byte of RAM per TOSSTI task than the default TinyOS scheduler. The code size increase for the static TOSSTI scheduler is also much 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 1.

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

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 2.

$$R(n) = T_0 + \sum_{i=1}^n T_i - \sum_{\alpha \in I} T_\alpha + \sum_{\alpha \in S} T_\alpha \quad (2)$$

### 3.3. Declaring and Posting Merged Tasks in nesC

The *TOSSTI* C preprocessor macro is provided to mark integrated tasks easily. To use it, 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

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

and all posts of the task in nesC become

```
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 *tids* 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.

Using a macro 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 section 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).

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.

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 = {ModA$task0, ModB$task1}
tasks03 = {ModA$task0, ModC$task3}
```

The first line indicates that an integrated version of *task0* of module *ModA* and *task1* of module *ModB* resides in the function named *tasks01*. 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.

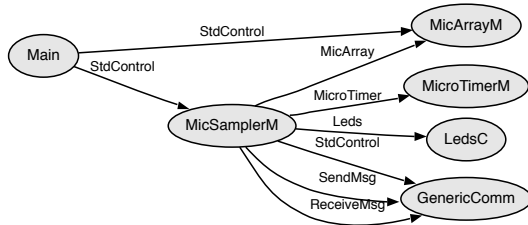
### 3.3.3. Building the TOSSTI Application

After the integrated tasks file is created the, TOSSTI script is used to insert an appropriate scheduler into the source code. The resulting code is compiled using gcc, along with the integrated function bodies, using the same compiler optimization flags that the nesC compiler passes to gcc, and then can be loaded onto the mote.

## 4. Experiments and Analysis

### 4.1. Sample TOSSTI 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 [17]. 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 for acoustically determining the locations of other motes in a network, or for aiding people with disabilities [1].

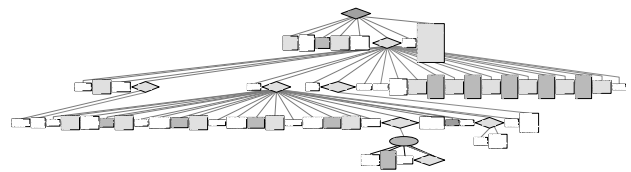


**Figure 6. Components used in MicSampler as generated using nesdoc**

When the mote boots, the radio is configured, the ADC is configured, and timer is configured to signal an interrupt periodically, at the desired sample rate. Whenever that interrupt occurs, the TinyOS signal handler for the timer 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, which have been integrated using STI. The transmit task is the standard TinyOS *startSend* task used by the micaZ platform. Figure 6 shows the nesC component diagram as generated by the nesdoc tool.

The sample and transmit tasks were integrated using *thrint*, a thread integration tool which can construct control dependence graphs (CDGs) of AVR assembly code and use those CDGs to integrate two tasks. Figure 7 shows the CDG of the *integrated* version of the sample and transmit tasks.



**Figure 7. CDG of Integrated getSamplesTask() and startSend()**

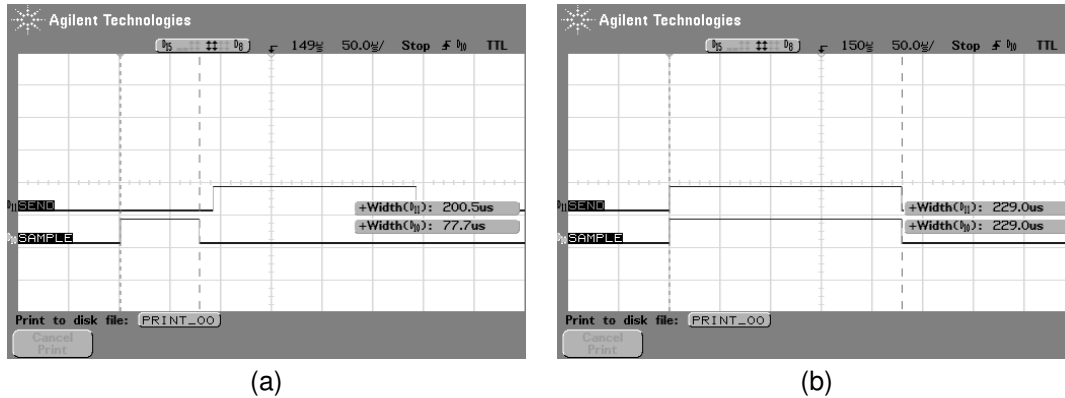
### 4.2. Analysis

Figure 8 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.

Figure 8a shows 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. Figure 8b 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.

#### 4.2.1. 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 in each ISR when the processor woke up from a sleep mode. The *TOSH\_run\_task()* function turned off that output pin before executing the AVR *sleep* instruction. Figure 9 also 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 pro-



**Figure 8. Oscilloscope screenshots showing task execution time (a) without and (b) with STI, using static scheduler**

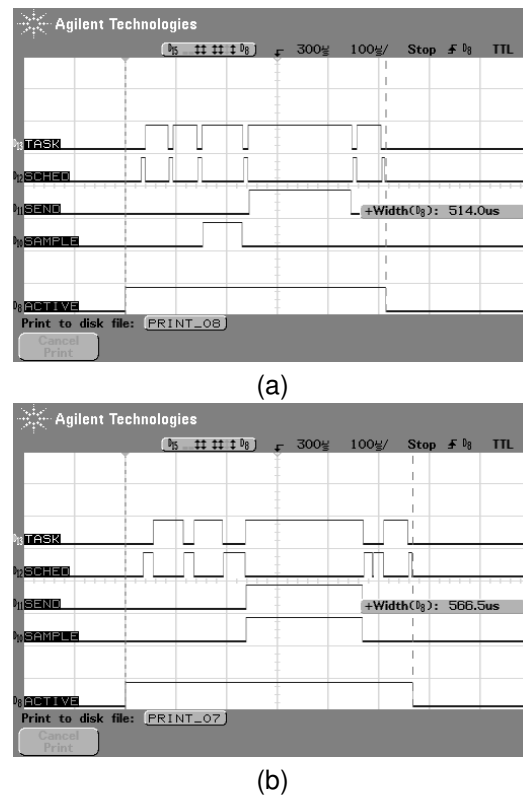
cessor: it is (c) 514.0  $\mu\text{s}$  when not using TOSSTI, and (d) 566.5  $\mu\text{s}$  when using TOSSTI: an *increase* in active time when using STI, not the expected *reduction*! When looking at how much time is spent actually spent searching for the next task to execute (waveform “SCHED”), (b) 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 3.2.2.

After the static TOSSTI scheduler was developed, the active-time analysis was done again. Figure 10 shows the active time of the same application using STI and the static TOSSTI scheduler is 481.5  $\mu\text{s}$ . When compared to the non-STI version of the application as seen in figure 9a, that is a time savings of 32.5  $\mu\text{s}$  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.

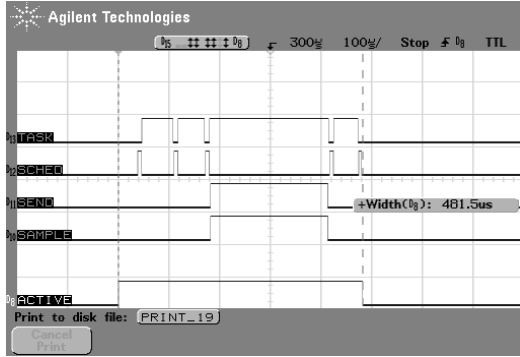
With other applications, including those that integrate security features or routing tasks, it is likely that the savings will be even greater [5][11][24]. Even though the runtime overhead using the dynamic TOSSTI scheduler is prohibitive for this application, in systems where new tasks may be introduced to the system at run time, the dynamic scheduler could be useful.

#### 4.2.2. Power and Energy

Based upon the experimentally measured active times presented the previous section, we now estimate power use.



**Figure 9. Oscilloscope screenshots showing active time (a) without and (b) with STI using dynamic TOSSTI scheduler**



**Figure 10. Oscilloscope screenshot showing MicSampler application using static TOSSTI scheduler**

We use the MicaZ power models presented by Polastre et al. [22]. We assume the processor is in standby mode when not waking up (180  $\mu$ s) or in active mode. We assume the radio is active for the same amount of time in each case to send the 12 bytes of data (source mote ID, packet number, and eight data samples) and 17 bytes of B-MAC protocol overhead. We assume a per-microphone sampling rate of 500 Hz (resulting in 500 messages per second).

We first examine MCU power for the application. Applying TOSSTI cuts processor active time by 6.3%, cutting average processor power by the same amount (from 7.54 mW to 7.19 mW).

Next we examine system (radio + MCU) power. TOSSTI cuts average system power from 38.92 mW to 38.57 mW. This 0.9% savings is modest because average system power consumption is dominated by the radio (average 31.4 mW), rather than the microcontroller (average 7.19 mW). The radio is used frequently in this application and has a large wake-up time. Furthermore, B-MAC protocol overhead extends the radio's active time. A platform with a more efficient radio or an application with more computation would see larger benefits from using TOSSTI.

#### 4.2.3. 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. Here, 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 2, the response time for *startSend* in the integrated versions of the application is now

$$R(startSend) = (T_{getSamplesTask} + T_{startSend}) - (T_{getSamplesTask} + T_{startSend}) + T_{sendAndSample}$$

#### 4.2.4. Program Memory Usage

Non-STI App	11,130 bytes
App with Dynamic TOSSTI Scheduler	13,166 bytes
App Static TOSSTI Scheduler	12,948 bytes
sendAndSample function	1,168 bytes

**Table 1. Program memory usage for versions of MicSampler application.**

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 1 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.

## 5. Conclusions

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. During what was previously busy-wait time, the mote can now perform useful operations, completing its active cycle sooner, going back a low-power mode sooner, reducing energy consumption. In the sample application, active time was reduced 6.3%. Applications with more idle time will benefit even more. 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 [16].

## References

- [1] Acoustic magic — product page. <http://www.acousticmagic.com/products/>, August 2005.

- [2] R. Braynard, A. Silberstein, and C. Ellis. Extending network lifetime using an automatically tuned energy-aware MAC protocol. In K. Romer, H. Karl, and F. Mattern, editors, *EWNS*, pages 244–259. LNCS, 2006.
- [3] M. Buettner, G. Yee, E. Anderson, and R. Han. X-MAC: A short preamble MAC protocol for duty-cycled wireless sensor networks. Technical Report CU-CS-1008-06, University of Colorado at Boulder, May 2006. <http://www.cs.colorado.edu/department/publications/reports/docs/CU-CS-1008-06.pdf>.
- [4] A. G. Dean. *Software Thread Integration for Hardware to Software Migration*. PhD thesis, Carnegie Mellon University, 2000.
- [5] A. G. Dean. Compiling for fine-grain concurrency: Planning and performing software thread integration. December 2002. <http://www.cesr.ncsu.edu/agdean/RTSS.02/Dean%20C4FGC%20Dist.pdf>.
- [6] A. G. Dean. Software thread integration research at NC State University. <http://www.cesr.ncsu.edu/projects/sti/websti/home.html>, March 2002.
- [7] A. Dunkels. The Contiki operating system, November 2004. <http://www.sics.se/contiki>.
- [8] A. Dunkels. Protothreads — lightweight, stackless threads in C, November 2004. <http://www.sics.se/~adam/pt>.
- [9] A. Dunkels, O. Schmidt, and T. Voigt. Using protothreads for sensor node programming. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, CO, November 2006. ACM. Also available at <http://www.sics.se/~adam/dunkels06/protothreads.pdf>.
- [10] D. G. et al. TinyOS tutorial, September 2003. <http://www.tinyos.net/tinyos-1.x/doc/tutorial/>.
- [11] P. Ganesan and A. G. Dean. Enhancing the AvrX kernel with efficient secure communication using software thread integration. In *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, Toronto, Canada, May 2004.
- [12] D. Gay, P. Levis, D. Culler, and E. Brewer. nesC 1.1 language reference manual, May 2003. <http://nesc.sourceforge.net/papers/nesc-ref.pdf>.
- [13] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *ACM SIGPLAN*, 2003.
- [14] R. Ghattas and A. G. Dean. Energy management for commodity short-bit-width microcontrollers. In *International Symposium on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. ACM, September 2005.
- [15] K. Klues, V. Handziski, C. Lu, A. Wolisz, D. Culler, D. Gay, and P. Levis. Integrating concurrency control and energy management in device drivers. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 251–264, New York, NY, USA, 2007. ACM.
- [16] P. Levis and C. Sharp. TEP106: Schedulers and tasks, February 2007. [http://tinyos.cvs.sourceforge.net/\\*checkout\\*/tinyos/tinyos-2.x/doc/html/tep106.html](http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep106.html).
- [17] K. Luthy. The development of textile based acoustic sensing arrays for sound source acquisition. Master's thesis, North Carolina State University, 2003.
- [18] U. Malesci and S. Madden. A measurement-based analysis of the interaction between network layers in TinyOS. In K. Romer, H. Karl, and F. Mattern, editors, *EWNS*, pages 292–309. LNCS, 2006.
- [19] D. Manjunath and R. Hansdah. A review of current operating systems for wireless sensor networks. [http://www.ece.iisc.ernet.in/network\\_labs/manjunath/commag.pdf](http://www.ece.iisc.ernet.in/network_labs/manjunath/commag.pdf), June 2006.
- [20] U. of Colorado at Boulder. MANTIS: HomePage, November 2004. <http://mantis.cs.colorado.edu/>.
- [21] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *Proceedings of the Second International Conference on Embedded Networked Sensor Systems (SenSys 2004)*, pages 95–107, Baltimore, Maryland, November 2004. ACM. Also available at <http://www.polastre.com/papers/sensys04-bmac.pdf>.
- [22] J. Polastre, R. Szewczyk, and D. Culler. Telos: enabling ultra-low power wireless research. *Fourth International Symposium on Information Processing in Sensor Networks, 2005. IPSN 2005.*, pages 364–369, 15 April 2005.
- [23] S. Roundy, P. K. Wright, and J. M. Babey. *Energy Scavenging for Wireless Sensor Networks*. Kluwer Academic Publishers, Boston, 2004.
- [24] R. Venugopalan and A. G. Dean. Improving energy efficiency in sensor networks by raising communication throughput using software thread integration. Center for Embedded Systems Research, North Carolina State University, March 2004.
- [25] B. Welch, S. Kanaujia, A. Seetharm, D. Thirumalai, and A. G. Dean. Extending STI for demanding hard-real-time systems. In *International Symposium on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*. ACM, 2003. Also available from <http://www.cesr.ncsu.edu/agdean/stiglitz/CASES03-STIGLitz-DHRTS.pdf>.