A Comprehensive Experimental Comparison of Event Driven and Multi-Threaded Sensor Node Operating Systems

Cormac Duffy[1], Utz Roedig[2], John Herbert[1], Cormac Sreenan[1]
[1]Computer Science Department, University College Cork, Ireland
Email: {c.duffy,j.herbert,c.sreenan}@cs.ucc.ie
[2]InfoLab21, Lancaster University, UK
Email: u.roedig@lancaster.ac.uk

Abstract— **The capabilities of a sensor network are strongly influenced by the operating system used on the sensor nodes. In general, two different sensor network operating system types are currently considered: *event driven* and *multi-threaded*. It is commonly assumed that event driven operating systems are more suited to sensor networks as they use less memory and processing resources. However, if factors other than resource usage are considered important, a multi-threaded system might be preferred. This paper compares the resource needs of multi-threaded and event driven sensor network operating systems. The resources considered are memory usage and power consumption. Additionally, the event handling capabilities of event driven and multi-threaded operating systems are analyzed and compared. The results presented in this paper show that for a number of application areas a thread-based sensor network operating system is feasible and preferable.**

Index Terms— **Sensor Networks, Sensor Network Operating Systems, Performance Evaluation, TinyOS, MANTIS.**

I. Introduction

Wireless sensor networks consist of battery powered sensor nodes used to gather information about a monitored physical phenomenon. To ensure long periods of unattended network operation, the energy consumption of the sensor nodes must be very low. The operating system used for the sensor nodes influences energy consumption on two levels. First, the design of the operating system defines the minimum resource requirements such as CPU speed and memory capacity that the sensor hardware must provide. Second, the operating system design influences the usage pattern of the CPU and thus defines how often energy-efficient sleep periods can be activated. As the sensor node must provide a service, the operating system design can not be focused solely on energy efficiency. Sensor readings and incoming messages must be handled promptly by the sensor node. The responsiveness of the node must be sufficient to handle events and requests according to their deadlines.

Application responsiveness is influenced by the operating system design in two ways. First, the responsiveness depends on the task scheduling capabilities of the operating system. Second, the design of the operating system influences the degree to which task scheduling is performed by the system or by the systems programmer. Obviously, the last point is subjective as it depends on the quality and knowledge of the programmers. Nevertheless, the issue has strong impact in real world application designs.

Currently, operating systems for sensor nodes follow either one of two different design concepts. Operating systems following the first design approach are called *event driven*. Each operating system process is triggered in response to an event (e.g. a timer, an interrupt indicating new sensor readings or an incoming radio packet). The tasks associated with the event are executed and thereafter the node is sent to an energy-efficient sleep state or the next event is processed. As events are processed sequentially, expensive context switching between tasks is not necessary. An example of such an operating system is TinyOS [1]. The second approach follows the classic *multi-threaded* operating system design. The operating system multiplexes execution time between the different tasks, implemented as threads. While switching from one thread to another, the current context has to be saved and the new context must be restored. This consumes costly resources in the constrained sensor node. An example of such an operating system for sensor nodes is MANTIS [2].

Currently it is assumed that an event driven operating system is more suitable for sensor networks because less resources are needed resulting in a more energy-efficient system. However, the exact figures are unknown and therefore determined and presented in this paper. On the other hand it is claimed that a multi-threaded operating system has superior event processing capabilities. Again, an in depth analysis is currently missing and is therefore shown in the paper. For comparison purposes, the event-based system *TinyOS* and the multi-threaded system *MANTIS* executing the same sensor network applications on the *DSYS25* [3] sensor platform are used. The contributions of the paper can be summarized as follows:

1) Comparison of the memory requirements of the event-based TinyOS and multi-threaded MANTIS operating system.

2) Comparison of the event processing capabilities of the event-based TinyOS and multi-threaded MANTIS operating system.
3) Comparison of the energy consumption of the event-based TinyOS and multi-threaded MANTIS operating system.

The results presented in the paper can be used to decide which type of operating system should be used for a specific sensor network application. The results also show that for a number of application areas a thread-based sensor network operating system is actually feasible and even preferable.

The rest of the paper is organized as follows. The next section describes related work in the research area of operating systems for sensor networks. Section III gives an overview of the operating systems TinyOS and MANTIS. Section IV describes the sensor platform, application scenarios and their implementation as used for the comparative experiments. Sections V,VII and VI present the experimental comparison of the operating systems. Section VIII concludes the paper.

II. Related Work

Historically, there has been much debate on whether an event-based or multi-threaded architecture is more efficient. To decide which operating system architecture should be used, the specific application environment of interest has to be considered as each application environment dictates very specific constraints. Unfortunately, results obtained for one application environment are normally not directly applicable to another environment. Thus, a comprehensive study covering a multitude of application scenarios is required.

Preliminary results of our research presented in this paper were published in [4]. This paper describes the performed experiments in much greater detail. In addition, the results and their implications are analyzed and discussed.

Besides our preliminary work, no published research exists that presents a comprehensive comparison of event-based and multi-thread sensor network operating systems taking event processing, energy consumption and memory usage into account. The lack of such a study is the main motivation for the work presented in this paper. Existing work targets only a subset of aspects investigated in this paper. For example papers analyzing or describing one specific operating system (e.g. [5], [6], [7], [8], [9]), or publications comparing only one aspect (e.g. memory usage in [6]). As each single existing analysis is based on different assumptions and experimental setups, it is not possible to extract an objective comparison.

As previously mentioned, the choice of operating system design has an impact on the way an application is programmed. In [10] the event-based system CONTIKI[7] uses a programming concept called "proto-threads" which allows the programmer to develop a program using a multi-threaded programming syntax. It is argued that an event-based system is more power efficient but that programming concurrent (sensor network) applications with threads as opposed to event handlers is easier for the programmer. The proto-threads in CONTIKI allow a combination of both benefits. However, an objective performance comparison of an event-based and multi-threaded system is not provided.

In [11] it is argued that TinyOS, in contrast to MANTIS, has a problem with multiplexing long running tasks and short running tasks. However, an in-depth analysis - an analytical evaluation or by measurement - of the task handling problem is not given. It is argued that both operating systems should be combined to overcome the existing problems. As a solution, the complete TinyOS operating system is executed as a thread in MANTIS. If tasks in TinyOS need to be able to preempt other tasks, they can be executed in a separate MANTIS thread. Very similar earlier work [12] uses the AvrX [13] as a multi-threaded extension of the TinyOS scheduler. In this work, the event processing capabilities of the basic TinyOS and the modified AvrX extended TinyOS are investigated (a very similar experimental setup to the setup described in Section IV is used for evaluation). However, an analysis of other parameters such as power consumption and memory usage is omitted. The differences of a generic thread-based system are also not investigated. In summary, the research focus of [11] and [12], is to integrate multi-threaded features in TinyOS.

In [14] TinyOS is compared with eCos, an embedded multi-threaded operating system. eCos is not specifically a sensor network operating system as many of the core operating features are designed for more complex embedded processors. The paper does evaluate both memory, processing performance and power efficiency. For processing performance the number of clock cycles required to process kernel functions are summed up, but actual application performance is not investigated. The power performance is compared theoretically based on the memory requirements of each operating system, but no investigation into operating system power-management is analyzed in this paper.

To our knowledge, this is the first work that provides specific data to determine the effectiveness of both an event-based and a multi-threaded operating system in a range of application scenarios.

III. Sensor Node Operating Systems

Two different sensor network operating system types are currently considered: *event driven* and *multi-threaded*. To compare both operating system concepts, a well known and widely used implementation of each was selected, namely *TinyOS* and *MANTIS*. The following paragraphs describe the basic functionality of each operating system, especially regarding power consumption and event processing which are the parameters of interest in our study. It has to be noted that the terms event, task and thread are used differently within the literature describing TinyOS and MANTIS. Thus, definitions are given below to avoid confusion:

- *Event:* An environmental occurrence.
- *Interrupt:* An interrupt triggered by an event.
- *Event-Handler:* A system function, invoked in response to an interrupt.
- *TinyOS Task:* A deferred procedure, usually triggered by an event-handler.
- *MANTIS Thread:* A portion of the MANTIS OS that can run independently of, and concurrently with, other portions of the OS, typically invoked in response to an interrupt.

```
 1: component_A
 2:   task do(){...}
 3:   command X(){...}
 4:   event Y(){...}

 5: int_A
 6:   ...
 7:   post_task(A)

 8: TOSH_run_task()
 9:   while(TOSH_run_next_task())
10:   TOSH_sleep()
```

Fig. 1. TinyOS structure

A. TinyOS

The TinyOS [1] operating system was one of the first operating systems specifically designed for wireless sensor networks. The inventors felt that an event driven light-weight kernel was the best solution to handling a "large number of concurrent flows and juggle numerous out-standing events".

The operating system and specialized applications are written in the programming language nesC and are organized in self-contained components. A simplified view of this component structure is shown in Fig. 1. Components consist of interfaces in the form of *command* and *event* functions. Components are assembled together, connecting interfaces used by components to interfaces provided by others, forming a customized sensor application. The resulting component architecture facilitates event-based processing by implementing event-handlers and TinyOS tasks. TinyOS tasks are deferred function calls and are placed in a simple FIFO task-queue for execution (see Fig. 1, line 8). TinyOS tasks are taken sequentially from the queue and are run to completion. Once running, the TinyOS task can not be interrupted (preempted) by another TinyOS task. Event-handlers are triggered in response to a hardware interrupt and are able to preempt the execution of a currently running TinyOS task (see Fig. 1, line 5). Event-handlers perform the minimum amount of processing to service the event. Further non time-critical processing is performed within a TinyOS task that is created by the event handler. After all TinyOS tasks in the task queue are executed, the TinyOS system enters a sleep state to conserve energy (see Fig. 1, line 10). The sleep state is terminated if an interrupt occurs.

Functionality in TinyOS is distributed among many components. Each function (e.g. sensing or packet forwarding) is normally segmented into a series of sub-steps, (task, command and event hander functions). Thus, functionality can execute concurrently by multiplexing these atomic sub-steps. When used correctly, this approach leads to an efficient system structure. On the other hand it is difficult for the programmer to achieve a proper division of functionality. The programmer must be familiar with the internal affairs of all low level components in the system. Additionally, it can be difficult for some programmers to handle functionality in such sub-steps.

B. MANTIS

The MANTIS operating system [2] is a light-weight multi-threaded operating system capable of multi-tasking processes on energy constrained distributed sensor networks. The system provides a classical multi threaded system in the context of wireless sensor networks.

Each task the operating system must support can be implemented - using standard C - as a separate MANTIS thread. A simplified view of this thread structure is shown in Fig. 2. A new thread is initialized and thread processing is started (line 1). Processing might be halted using the function *mos_semaphore_wait* when a thread has to wait for a resource to become available (line 3). An interrupt handler (line 4) using the function *mos_semaphore_post* (line 5) is used to signal the waiting thread that the resource is now available and thread processing is resumed. While a thread is waiting on a resource to become available, other threads might be activated or, if no other processing is required, a power saving mode is entered. Power saving is handled by a thread called idle-task which is scheduled when no other threads are active. Thread scheduling is performed within the kernel function *dispatch_thread* shown in Fig. 2, line 6. This function searches a data structure called *readyQ* for the highest prioritized thread and activates it. When the *dispatch_thread* function is called, the current active thread is suspended calling *PUSH_THREAD_STACK* (line 7) which saves CPU register information. The highest priority thread is then selected from the *readyQ* (line 8) and its register values are restored by the *POP_THREAD_STACK* function (line 10). Before the *dispatch_thread* function is called, the *readyQ* structure is updated. Threads that are currently sleeping or that are waiting on a semaphore (resource) are excluded from the *readyQ*. The scheduling through the *dispatch_thread* function can be initiated by two different means. *Dispatch_thread* is called when a semaphore operation is called (e.g. to let the current thread wait on a resource). *Dispatch_thread* is also called periodically by a time slice timer to ensure processing of all threads according to their priority.

Like all multi-threaded operating systems, MANTIS was developed with a complement of built-in memory protection techniques such as binary and counting semaphores to manage and coordinate threads. MANTIS threads are implemented in the MANTIS kernel with a static thread table, which stored priority and state

```
thread_A
  while(running)
     ...;mos_semaphore_wait(A1);...

int_A
  ...;mos_semaphore_post(A1);...

dispatch_thread()
  PUSH_THREAD_STACK()
  CURRENT_THREAD = readyQ.getThread()
  CURRENT_THREAD.state=RUNNING
  POP_THREAD_STACK()
```

Fig. 2. MANTIS structure

information for each MANTIS thread and a pointer to the thread stack. Typically only 12 MANTIS threads can be either queued or active at any one time and will consume 120 bytes of SRAM [2]. The heap space of each thread is managed by the kernel memory manager. The memory manager assigns thread space according to a best-fit policy. Thread heaps will, by default, be assigned in the lowest possible memory address spaces, ensuring that a thread heap space doesn't collide with the processor heap.

By defining processes as threads, algorithms can be represented in a more intuitive sequential fashion as processes don't need to be segmented into specific states when waiting for a system event to occur. Compared to TinyOS, functionality is multiplexed by the OS, not by the programmer. Thus, MANTIS has the potential of being easier for programmers to use.

IV. Evaluation Setup

For the evaluation of event processing capabilities, power consumption patterns and memory usage of TinyOS and MANTIS, both operating systems were ported to the same sensor platform, the DSYS25 [3]. Additionally, measurement facilities are integrated in both operating systems which allow us to observe the required parameters without altering the system behavior. To actually perform the comparative evaluation, an abstract application scenario is defined and implemented on the sensor nodes. This abstract application scenario corresponds to a broad spectrum of real-world sensor network deployments.

In the following paragraphs, the selected abstract application scenario is motivated and described. Thereafter, the DSYS25 sensor-node platform, the ported operating systems and the measurement hooks are explained. This evaluation setup is used for the experiments described in the remaining sections of the paper.

A. Application Scenarios

To evaluate operating system performance, an application context must be defined. Subsequently, the operating system performance of a sensor node supporting the given scenario can be investigated. Obviously, to obtain useful results, it is important that the investigated application scenario corresponds to real-world deployment and usage scenarios of wireless sensor networks.

In many cases, a sensor network is used to collect periodically obtained measurement data at a central point (sink or base-station) for further analysis. The sensor nodes in such a network perform two major tasks. Sensor nodes perform the sensing task and they are used to forward the gathered data to the sink. If the sink is not in direct radio range of a node, other nodes closer to the sink are used to forward data. The execution time of the sensing task will depend on the nature of the physical phenomenon monitored and the complexity of the algorithm used to analyze it. Therefore, the position of the node in such a network and the complexity of the sensing task define the operating system load of the sensor node. The complexity of the sensing task is varied in the experiments and hence the application scenario is considered abstract, as it can be compared with many different real-world deployment scenarios.

The complexity of the sensing operation depends on the phenomenon monitored, the sensor device used and the data pre-processing required. As a result, the operating system can be stressed very differently. If, for example, an ATMEGA128 CPU with a processing speed of $4Mhz$ is considered (a currently popular choice for sensor nodes), a simple temperature sensing task (processed through the Analogue to Digital Converter) can be performed in less than $1ms$ [15][1]. In this case only a $16bit$ value has to be transferred from the sensing device to the CPU. If the same device is used in conjunction with a camera, image processing might take some time before a decision is made. Depending on camera resolution and image processing performed, a sensing task can easily take more than $100ms$ [16]. Other application examples documented in the literature are situated in between these values. For example if a sensor value needs to be cryptographically secured before transmission, the sensing task is prolonged by $5ms$ [17]. Thus, combined with a simple sensor, a sensing task can be around $10ms$. Note that a long sensing task can be split-up into several sub-tasks. However, in practice this is often not possible due to two factors. First, many data-processing algorithms are difficult or impossible to be split into separate tasks. For example some image processing algorithms cannot be divided, as explained in [16]. Second, it is difficult from a programming (or programmers) perspective to handle and manage such divided entities. The experimental evaluation spans the task sizes described ($1ms ... 100ms$). Thus, the abstract application scenario corresponds to a wide range of real world application scenarios.

The following paragraphs give an exact specification of the abstract application scenario used, which is defined by its topology, traffic pattern and sensing pattern. The application scenario is then implemented on the DSYS25 sensor platform for evaluation.

[1]This calculation is based on the amount of processing time necessary to process an analog sensor reading taken from the Atmega ADC. The ADC is typically used to process signals from analog sensors such as a temperature sensor.

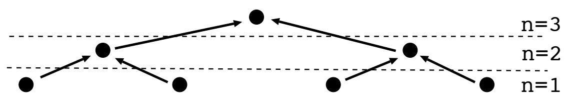


Fig. 3. Binary Tree

Topology: The sensor network is used to forward sensor data towards a single base-station in the network. It is assumed that a tree topology is formed in the network. To simplify the evaluation process, a binary tree shown in Fig. 3 is assumed. Depending on the position n in the tree, a sensor node s_i might process varying amounts of packets. Leaf nodes put less demand on the processor. Nodes closer to the root are more involved in packet forwarding and these nodes have to multiplex packet forwarding operations with their sensing operations. The position in the tree has therefore - besides other parameters - a significant impact on the event processing and energy consumption properties of the node.

Sensing Pattern: A homogeneous activity in the sensor field is assumed for the abstract application scenario. Each sensor gathers data with a fixed frequency f_s. Thus, every $t_s = 1/f_s$ a sensing task of the duration l_s has to be processed. As mentioned, the duration l_s is variable between $l_s = 4000$ and $l_s = 400000$ clock cycles depending on the type of sensing task under consideration (Which corresponds to $1ms/100ms$ on a $4MHz$ CPU).

Traffic Pattern: Depending on the position n of a node s_i in the tree, varying amounts of forwarding tasks have to be performed. It is assumed that no time synchronization among the sensors in the network exist. Thus, even if each sensor produces data with a fixed frequency, data forwarding tasks are not created at fixed points in time. The arrival rate λ_n of packets at a node at tree-level n is modeled as a Poisson process. As the packet forwarding activity is related to the sensing activity in the field, λ_n is given by:

$$\lambda_n = (2^n - 1) \cdot f_s \tag{1}$$

This equation is a simplification; queuing effects and losses are neglected, but nevertheless provides an accurate method to scale the processing performance requirements of a sensor network application. It is assumed that the duration (complexity) l_p of the packet forwarding task, is $l_p = 4000$ clock cycles. This is the effort necessary to read a packet from the transceiver, perform routing and re-send the packet over the transceiver. This is a common processing time and was obtained analyzing the DSYS25 sensor nodes using the Nordic radio [18].

Summary: The abstract scenario described above is used in Section V, Section VI and Section VII to compare event processing capabilities, energy consumption and memory usage of the two operating systems. For the evaluation, a single DSYS25 sensor node is fitted with the operating systems under investigation and the packet forwarding tasks and sensing tasks are generated such that the nodes activity corresponds to a place in the tree topology. The parameters defining the abstract application are set for all following experiments according to the values or ranges listed in Table I (Set for the $4MHz$ CPU used in the DSYS25 platform).

TABLE I
EVALUATION SETUP

Task	Duration	Clock cycles	Frequency
Forwarding	Fixed: $1ms$	Fixed: $l_p = 4000$	Variable, Poisson: $\lambda_n = (2^n - 1) \cdot f$ $n \in 1, ..., 8$
Sensing	Variable: $100ms$, $75ms$, $50ms$, $10ms$, $5ms$, $1ms$	Variable: $l_s = 400000$, $l_s = 300000$, $l_s = 200000$, $l_s = 40000$ $l_s = 20000$, $l_s = 4000$	Fixed, Periodic: $f_s = 1\frac{1}{s}$

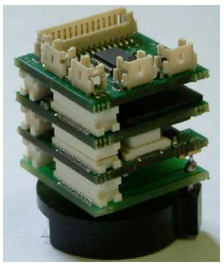


Fig. 4. DSYS25 Sensor Hardware

B. Evaluation Platform

The DSYS25 [3] is a sensor platform developed as part of the D-Systems project at University College Cork and the Tyndall Institute. The hardware platform is an ATMega128L micro-controller based Lego-like 25mm x 25mm stackable system. Its modular nature lends itself to the development of numerous layers for use in different application scenarios. Layers can be combined in an innovative plug and play fashion and include communication, processing, sensing and power. The sensor hardware is depicted in Fig. 4.

The chosen Nordic nRF2401 transceiver performs communication tasks such as address and CRC computation, freeing the micro-controller from these activities. Thus the micro-controller can be either used for processing other tasks or can be sent to an energy saving sleep mode.

C. OS Ports and Measurement Facilities

The goal of the study is a comparison of operating system concepts. To provide a fair comparison our experiment hinges on a number of important setup/experimental parameters.

Firstly, conceptual differences rather than functional differences between the two operating systems must be measured. For example the networking stack used in TinyOS and MANTIS provide different functionality (e.g MAC protocols, duty cycles ....). Thus, a direct comparison of the operating systems might be caused by functionality unrelated to the operating system. To avoid this problem, the operating systems are reduced to their bare minimum and simple components emulating the behavior of the aforementioned abstract application are implemented.

Secondly, in measuring the performance of each application it is imperative that the performance of the abstract application is not hindered. The experiment revolves around a setup in which tasks are dispatched up until the end time of the experiment. Performance is evaluated as to how timely each task is completed with respect to this clock. However in measuring clock cycles we take away processing resources from the tasks and would therefore provide imprecise measurements. This overhead is carefully calculated with an additional hardware timer and subtracted from the results.

TinyOS: In case of TinyOS, the abstract application is implemented using nesC, the programming language used in TinyOS. The packet processing task and the sensing tasks are initiated by an interrupt. For the experiment, the interrupt is not generated by the transceiver or sensor hardware, instead, the interrupt is generated by a timer. The timer intervals are configured by the parameters given for the abstract application (λ_n, f_s, see Table I) . Within the interrupt routine for the transceiver, a TinyOS task for packet forwarding is created and queued in the task list. The size of the packet forwarding task l_p is set to 4000 clock cycles (implemented as assembler NOOP operations, to ensure cycle accurate timings). In the interrupt routine of the sensor, the sensing task is created. The sensing task has the size (complexity) l_s defined by the application scenario.

The TinyOS is modified such that essential parameters can be measured during the experiment. The task creation time in the interrupt routine is recorded. When a task finishes, the task duration from creation to completion is known. Additionally, idle times of the CPU are recorded. The time from the completion of the last task to the occurrence of a new event is measured. The records of these two parameters - task execution time and system idle time - enables us to analyze event processing and power consumption capabilities of the operating system.

MANTIS: The thread-based architecture of the MANTIS operating system requires a syntactically different implementation of the abstract application. The sensing task and the packet forwarding task are implemented as MANTIS threads. Again, the interrupts initiating packet processing and sensing are generated by a timer, not the hardware. The timer intervals are configured by the parameters given for the abstract application (λ_n, f_s, see Table I). Following an interrupt, the appropriate interrupt routine is called. Within the interrupt routine, the necessary thread for processing is activated. MANTIS allows for a prioritization of threads. The packet processing thread is configured to have a higher priority than the sensing thread. The size of the packet forwarding thread l_p is set to 4000 clock cycles The sensing thread has the size (complexity) l_s defined by the application scenario. When a thread completes execution, it is set to sleep and waits to be woken again by the appropriate interrupt.

The time from waking a thread until its completion is measured during the experiments. Thus, an analysis of the event-processing capabilities of MANTIS is possible. If no thread is active, MANTIS activates an idle thread. The idle thread is used to implement the power management capabilities of the operating system. The idle thread decides which power saving mode has to be activated. Power saving is terminated when an interrupt occurs. The time from activating the idle thread until an interrupt occurs is measured. Thus, the power management capabilities of MANTIS can be investigated.

The following Sections V, VI, VII detail how each performance result was measured and the findings and relevance of these results.

TABLE II
MEMORY USAGE

Operating System	Programmable Flash Memory (kB)	Required RAM (B)
TinyOS	9	283
MANTIS	13.1	287

V. MEMORY USAGE

The memory footprint of the operating system has to be as small as possible. Additional memory increases the cost of the sensor nodes and the more memory is integrated in the hardware, the more energy is consumed by the system. In practice, not each reduction of memory requirements leads to a cheaper and more energy-efficient system. A sensor built out of standard components normally uses a chip combining CPU and memory. Whether the system uses the available memory or not has little impact on the power consumption of the chip. Thus, improvement can only be expected if memory (and CPU) requirements can be reduced to a point where a simpler CPU/memory chip is available.

A. Measuring memory

In order to determine operating system memory usage, we use the GNU project binary utility avr-size [19]. Avr-size is a flash image reader that outputs the program size, and the initialized and uninitialized memory size.

B. TinyOS

The TinyOS operating system core consists of an absolute minimum of functionality. Simple algorithms such as the FIFO queuing algorithm are used to implement core TinyOS features such as the task scheduler, in order to maintain compatibility with very limited processors. A structure is provided to multiplex many timed events with a single hardware timer (see Section III). All additional functionality is provided by the application code in the form of a component-based architecture. The TinyOS core elements alone do not form a useful system. Thus, it is necessary to analyze the memory usage of the TinyOS operation system combined with specific application code.

Operating system and application code are compiled into a single binary file which is executed on the sensor node. Depending on the application used, the executable requires an extremely small memory space. The abstract application scenario presented in Section IV is used for the evaluation.

Memory requirements of the OS/Application are further reduced by a specialized custom compiler (nesC) provided with the TinyOS framework. The nesC compiler exploits the component-based architecture to include only components required by the application's wiring schema in the compiled program image. The nesC compiler can further deduce and remove any unused component functions within the application configuration [20]. A TinyOS program image therefore contains no program code unrelated to the target application.

The necessary memory space for TinyOS providing the abstract application functionality is shown in Table II. The programmable flash memory, represents the amount of space to store the application code. The RAM field represents the amount of statically compiled memory/variables required by the application at runtime. Both applications will require more RAM to assign memory space for local variables which are dynamically allocated at run-time[2].

C. MANTIS

MANTIS OS provides more core functionality. The operating system core must provide functionality to handle multiple threads. Furthermore, preemption and context switching must be implemented within the operating system core. This includes semaphores, timer structures and memory management. Semaphores must be implemented to ensure that critical sections of code cannot be used by multiple threads at the same time. A memory manager must be set up to dynamically assign S-RAM address space during thread initialization. Three different hardware timers are required to implement time-slicing, general purpose timers and a sleep timer. As such the complexity of the MANTIS kernel requires considerable overhead in code size, memory requirements and processor complexity. An application is implemented as a set of MANTIS threads. Kernel and application are compiled into one binary file executed on the CPU. To be able to compare the memory footprint of MANTIS with TinyOS, the abstract application scenario is used for evaluation.

The MANTIS OS framework does not provide any tools to optimize memory size at compile time. Thus, it is left to the programmer to decide which operating system elements should be included in the binary. In practice, it is very difficult to determine which elements are necessary and a tool - comparable to nesC in TinyOS - would be very useful.

The necessary memory space for MANTIS to provide the abstract application functionality is shown in Table II.

[2]By dynamic memory, we refer to memory assigned by a stack, not the more common term of heap memory allocation.

D. Discussion and Findings

The initial build of each operating system highlights the huge memory savings gained by using TinyOS in combination with the nesC compiler. By compiling only the necessary functionality the nesC compiler can build extremely optimized binary images. A direct memory comparison between the operating systems is not fair if no optimization is performed for MANTIS. Thus, the MANTIS operating system was optimized manually for the experiment by removing all non-essential functionality. The results are shown in Table II. With this optimization, the MANTIS operating system takes nearly a third extra programmable memory space the TinyOS operating system. Both operating systems statically allocate almost equal amounts of RAM, however both applications will require more memory to cater for the stack (local/non static memory). Furthermore the MANTIS scheduler dynamically allocates a memory pool to store the stack and processor registers for each thread. If there is insufficient memory to store both the threads and stack, then during the course of execution, memory will be corrupted and the application will fail.

The experimental results show that an optimization method, comparable to the nesC features in TinyOS, would be desirable for the MANTIS operating system. If a standard Atmel 128 processor is used to realize such a system, the applications realized with either operating system can be accommodated easily in the memory[3]. If a more constrained platform is selected, such as the PIC 16 micro processor [21], a TinyOS implementation might be the only option to realize such a system.

VI. Event Processing

Many monitoring tasks in sensor networks require a responsive network reaction as sensing data has to be reported in a timely fashion. If the operating system of a sensor node is not capable of quickly responding to events, such time-critical applications are difficult to build. Furthermore, it is desirable to have sensor nodes that react to events in a deterministic and constant way. Sensor nodes with a predictable and constant performance can be used as building blocks for sensor network applications that require more deterministic network behavior.

This section investigates the event processing capabilities of the TinyOS and MANTIS operating system while supporting the abstract application scenario (see Section IV-A). The average processing time required to handle the packet forwarding task in the abstract application scenario is investigated. Additionally, the standard deviation of this processing time is investigated to determine the stability of processing times.

A. TinyOS

The simple TinyOS scheduler schedules tasks to run atomically with a FIFO queuing algorithm. Tasks will

[3]The Atmega 128 is capable of supporting 128kB of programmable flash memory, and 4096B of Static Random Access Memory(SRAM) [15].

execute atomically with respect to one another, and can only be preempted by an asynchronous event (an event spawned from a hardware interrupt). The scheduling algorithm is very simple, and it can therefore schedule tasks with a minimum processing overhead. By scheduling tasks to run atomically, TinyOS precludes any potential deadlock errors or any potential inter-task race conditions, greatly simplifying the task of programming the application. However, atomic/non-preemptive task schedules can result in non-ideal situations. A long TinyOS task (e.g. a sensing task) is occupying the CPU. Periodic interrupts occur that signal the arrival of packets. The packets are read from the radio in the interrupt routine (interrupting the long TinyOS task) and a TinyOS task for the packet processing is created to be processed after the long TinyOS task. Packet processing is deferred in a non-predictable way and deadlines regarding packet processing can be missed. Thus, if an application requires more precise control of the task execution schedule, the existing TinyOS scheduler is not suitable.

The task-blocking problem can be tackled with a number of different solutions. The functionality of the high-priority tasks can always be programmed into an interrupt handler. However code executed in an interrupt handler will block all other activity and should therefore be used sparingly. A more viable solution would be to segment the long sensing task into a series of sub-tasks, such that higher priority TinyOS tasks will not be blocked for the full duration of the long task execution time. However, as described in Section IV-A the segmentation of processing functions is often not possible or simply not done by the programmer.

B. MANTIS

In the multi-threaded MANTIS operating system, all processes are defined as individual MANTIS threads which can be preempted at any time during the course of execution. MANTIS thread preemption provides a significant scheduling advantage, as all higher priority tasks can be executed on demand. Thus, it is easier to ensure that deadlines of high priority tasks are met. However, MANTIS thread preemption facilitates extra processing cost, as the MANTIS scheduler must swap active/idle threads in addition to managing a memory pool to store the state of inactive threads. These extra costs add to the overall processing time needed to realize the task functionality. In the MANTIS operating system the processor will consume approximately 1200 clock cycles from the time the interrupt occurs until the time the tasks starts executing. The significance of this overhead depends on the length of each process and the number of context switches.

A long sensing task, implemented as low priority MANTIS thread can be interrupted for packet processing. Packets arrive at the radio triggering interrupts. An interrupt causes a high priority MANTIS thread to be activated for packet forwarding. As context switch time and packet forwarding have deterministic time bounds, deadlines regarding packet processing can be met.

C. Measuring Event Processing Capabilities

To ascertain the operating system responsiveness, experiments with the network model and experiment setup shown in Section IV-A are carried out. Two tasks, a sensing task and a packet processing task are executed concurrently on the same processor. In this model, the (lengthily) sensing task and the packet-processing task compete for CPU resources on the sensor node. It is assumed that the packet-processing task within the nodes has priority so that deadlines regarding packet forwarding can be met. Thus, in the MANTIS implementation, the packet processing task has a higher priority than the sensing task. In the TinyOS implementation, no prioritization is implemented as this feature is not provided by the operating system.

Task Execution Time: To characterize processing performance of the operating system, the average task execution time E_t of the packet forwarding task, is measured. During the experiment, J number of packet processing times e_j are recorded. To do so, the task start time e_{start} and the task completion time e_{stop} are measured and the packet processing time is recorded as $e = e_{stop} - e_{start}$. In case TinyOS is used, e_{stop} is the time when the packet processing task is completed and is removed from the task queue. When MANTIS is used, e_{stop} is the time when the packet processing thread finishes and is sent into the wait state. In both cases, the start time e_{start} is recorded in the interrupt routine when the packet processing is initiated. The average task execution time E_t is calculated at the end of the experiment as:

$$E_t = \frac{\sum e_j}{J} \tag{2}$$

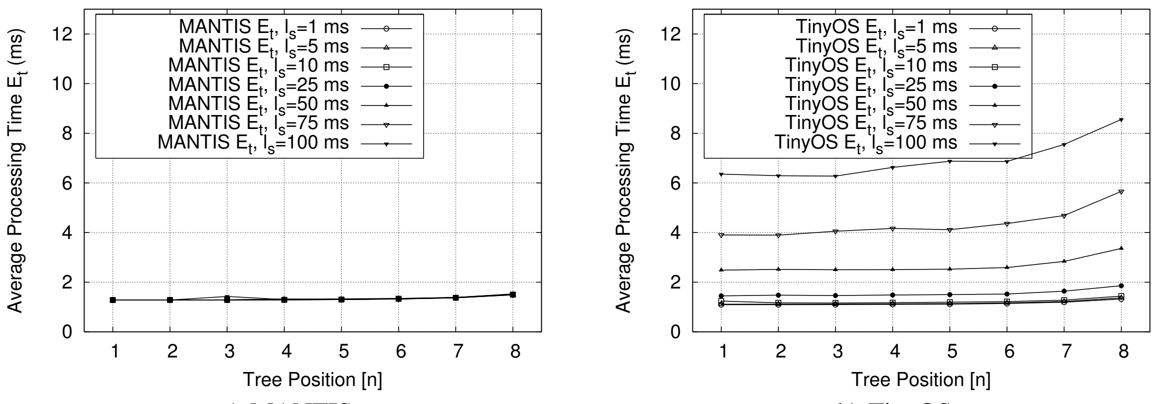
In order to investigate how deterministic the packet processing time is, the standard deviation of the average execution time is also calculated.

D. Evaluation

In the experiment, the average task execution time E_t is determined for TinyOS and MANTIS supporting the abstract application scenario. The average task execution time E_t is shown in Fig. 5.

Where MANTIS is used, it can be observed that the average packet processing time is independent of the sensing task execution time. Furthermore, E_t is also very independent from the position n of the node in the tree. Only under heavy load, the average processing time slightly increases. This is due to the fact that under heavy load packet forwarding tasks have to be queued (see Fig. 5 a)).

Where TinyOS is used, the average processing time for the packet forwarding task E_t depends on the length of the sensing l_s of the sensing task. In addition, under heavy load the queuing effects of the packet forwarding



a) MANTIS b) TinyOS

Fig. 5. Average packet processing time E_t , scheduled with a sensing task of execution time l_s.

tasks also contribute somewhat to the average processing time (see Fig. 5 b)).

The thread prioritization capability of MANTIS is clearly visible in the experiment results. Packet processing times are independent of the concurrently executed and lower priority sensing task. In TinyOS, sensing and packet forwarding task delays are coupled, and the influence of the sensing activity on the packet forwarding activity is clearly visible.

In the case of small sensing tasks with $l_s \leq 5ms$, TinyOS outperforms MANTIS. MANTIS has to perform a context switch (either from the idle task or a running sensing task) and this overhead adds to the average execution time E_t.

Fig. 6 shows the standard deviation of the packet forwarding task execution time E_t for a sensing task of length $l_s = 75ms$. In the case where MANTIS is used, the standard deviation in the task processing time is very low. The standard deviation is noticed only for a high system load. This is caused by packets queuing and waiting to be processed. If TinyOS is used, a huge variation in the processing times is observed. This variation increases considerably with the size of the sensing task. This is due to the fact that a packet processing task in TinyOS has to wait for a sensing task to finish. However, for small sensing tasks, as shown in Fig. 6, the standard deviation of E_t is very small.

E. Discussion and Findings

It is evident that MANTIS provides better stability and predictability of packet processing time than TinyOS. Additionally, MANTIS decouples the packet processing time from the sensing time delay.

However, if small sensing tasks are used, TinyOS is able to provide a similar stability in the packet processing times. In fact, in the case of small processing times, TinyOS is capable of processing incoming traffic faster as there is no operating system overhead in the form of context switches.

If a sensor network with a controlled and predictable network performance has to be implemented, a deterministic packet forwarding behavior of sensor nodes is required. For such an application, the MANTIS operating system would generally be more suitable, especially if processing intensive sensing tasks have to be supported concurrently. However, in the particular case of small sensing tasks, TinyOS is the better choice. The same stability in packet processing time can be achieved while packet processing requires less time. Thus, a higher throughput can be achieved.

Our findings regarding event processing of event-based and multi-threaded sensor network operating systems can be summarized as follows:

- A multi-threaded system is preferred if long (sensing) tasks have to be supported concurrently with the packet forwarding tasks.
- An event-based system is preferred if short (sensing) tasks have to be supported concurrently with the packet forwarding tasks.

While these conclusions are not surprising, they nevertheless provide quantitative results on which to support the choice of operating system.

VII. Energy Consumption

The lifetime of a sensor network is related to the energy consumption of the sensor nodes. Therefore, the task of reducing a sensor node's power consumption is of paramount importance. Operating systems for sensor networks achieve low power consumption rates by exploiting processor idle times. Available idle time can be used to put the CPU in an energy-efficient power saving mode. Depending on the processor used, different power saving modes might be available. The different power saving modes vary in the time and energy necessary to enter and leave the mode and the power spent in the particular mode. Thus, to determine the energy efficiency of sensor network operating systems, it is necessary to investigate the available idle times during system operation.

This Section investigates the available idle time in TinyOS and MANTIS while supporting the abstract application scenario (see Section IV-A). First, the ratio between idle and active time is determined as this number dictates how much energy can possibly be saved. Second, the

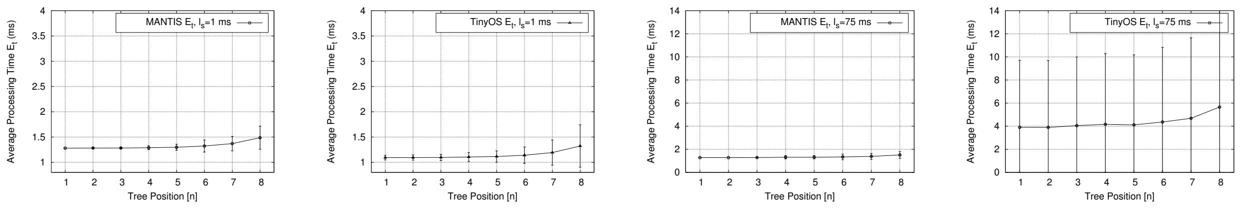


Fig. 6. Standard deviation of packet processing time E_t, scheduled with a sensing task of execution time$l_s = 1ms$ and $l_s = 75ms$.

distribution of idle time periods is analyzed as the length and variation of available idle periods determines which power saving mode can be used by the processor.

A. TinyOS

TinyOS executes all pending tasks sequentially (See Section III). A processing task is only interrupted by hardware interrupts which in turn process interrupt service routines. Thus, the TinyOS operating system spends almost all processing activity in the execution of the application functionality. Little processing effort is spent on operating system related functionality.

As soon as the task queue is emptied and no events have to be processed, the TOSH_sleep() routine is called. In this routine it can be decided how the CPU should spend the available idle time. More specifically, it can be decided which power saving mode will be used. Here it is theoretically possible to use an algorithm to predict how long the idle period will be and to select the power saving mode accordingly. Such an algorithm might take application layer knowledge into consideration. However, the current implementation of TinyOS only considers the simplest available power saving mode and optimization options are not exploited.

B. MANTIS

The drawback of a multi-threaded operating system, is that a considerable part of the CPU processing time is needed for the organization of the system itself. A context switch between different threads requires the operating system to save the context information. This operating system overhead can not be spent for power efficient sleep times. The question is how much overhead has to be allocated for the operating system itself.

If no MANTIS threads are scheduled for execution and no events have to be processed, the system executes a so-called idle-task. The function of the idle-task is to determine which power saving mode should be activated. For this decision, the idle-task uses thread state information. If all threads are waiting for a service to respond, i.e the radio to return an acknowledgment message, then it is in a THREAD_STATE_IDLE and the idle-task will activate a CPU idle mode as soon as all threads finish processing. If however, all threads are in a THREAD_STATE_SLEEP then there are no threads waiting on a service to complete, and the CPU will enter a sleep mode once all threads finish processing. This simple policy is used as it is assumed that a thread in THREAD_STATE_IDLE will become active very soon and no long idle period can be expected. The idle mode can be activated without a transition phase but is not very energy-efficient. The sleep mode has a transition phase before the actual power saving starts. This simple policy helps to optimize power consumption and depends very much on the type and specification of the CPU used. However, a better policy might be available if, for example, application layer knowledge is used to predict idle times.

C. Measuring Power Efficiency

To measure the power efficiency of each operating system, we use the abstract application scenario in Section IV-A, to evaluate each operating system under varying degrees of duress. Both operating system applications must process a sensing task and a number of packet forwarding tasks. The frequency of packet forwarding tasks increases progressively as the position of the sensor node reaches the root of the network topology, i.e. the number of child nodes increase.

In evaluating power efficiency, the aforementioned operating system specific power management policies are ignored (In fact, only MANTIS currently implements such a policy). This study investigates the available idle times for power management purposes, not the different power management policies available. An efficient power management policy must be tailored to the particular CPU used and can take application layer information into account. However, the results presented can be used to design appropriate power management strategies.

Idle time: The first parameter measured is the percentage idle time. The percentage idle time indicates how power efficient the system can be. The longer the idle time the more energy-efficient the system. In the experiment, the abstract application scenario is executed by the sensor node running TinyOS or MANTIS. The duration of the experiment T and the duration i_k of k idle time periods during the experiment is recorded. i is defined as $i = i_{stop} - i_{start}$. In case of TinyOS, i_{start} is the point in time when to TOSH_sleep() is called. In case of MANTIS, i_{start} is defined as the point in time when the idle-task begins execution. i_{stop} is for TinyOS and MANTIS the point in time when the system resumes operation by processing an interrupt. All idle periods i_k are summarized and the percentage idle time, I_t, the percent of experiment time, in which the processor is idle, which is calculated as follows:

$$I_t = \frac{\sum i_k}{T} \cdot 100 \qquad (3)$$

The percentage idle time is compared with the theoretical maximal percentage idle time, I_k^{max}. I_k^{max} is calculated by taking only application processing of the abstract application scenario into account. Thus, I_k^{max} represents the percent of running time the processor would be idle for an ideal operating system which would have no operating system processing overhead. I_k^{max} depends on the task sizes l_s and l_p of sensing and packet forwarding task, the frequency of the sensing task f_s, the CPU speed s_{cpu} and the position n of the node in the abstract application scenario. I_k^{max} is calculated using Equation (1):

$$I_k^{max} = \left(1 - \frac{f_s}{s_{cpu}} \cdot (l_s + l_p \cdot (2^n - 1))\right) \cdot 100 \qquad (4)$$

The operating system overhead I_k^o is calculated using Equation (3) and Equation (4):

$$I_k^o = I_k^{max} - I_t \qquad (5)$$

Idle Periods: The second parameter of interest is the average length of the idle periods I_p. This parameter shows which sleep modes a CPU could use. Long continuous idle periods allow for deep sleep modes. Again, it is necessary to record the duration i_k of all K idle time periods during the experiment. The average length of the idle period I_p is calculated as:

$$I_p = \frac{\sum i_k}{k} \qquad (6)$$

The standard deviation of the idle period length is also calculated to determine the stability of the idle period lengths.

D. Evaluation

In the first experiment, the percentage idle time I_t is determined for TinyOS and MANTIS supporting the abstract application scenario. The idle time I_t is shown in Fig. 7.

The time spent in idle mode drops for both operating systems exponentially with the increasing node position in the tree described by the parameter n. This behavior is expected as the number of packet tasks increases accordingly (See Equation (1)). Less obvious is the fact that the available idle time drops faster in MANTIS than in TinyOS. The fast drop in idle time is caused by the context switches in the MANTIS operating system. The more packet forwarding tasks are created, the more likely it is that a sensing task is currently running when a packet interrupt occurs. Subsequently, a context switch to the higher prioritized forwarding task is needed.

When MANTIS is used, the length of the sensing task has a significant impact on the idle time. If TinyOS is used, the length of the sensing task does not influence idle time that strongly. Again, the difference is down to the necessary context switches in MANTIS. The longer the sensing task, the more likely it is that a sensing task is running when a packet arrives.

As expected, MANTIS proves to be less energy-efficient than TinyOS. However, in the case of low system activity both systems have roughly the same energy efficiency. For example for a leaf node with $n = 1$ and a sensing task with the size of $l_s = 1ms$ TinyOS is only 0.09% more energy-efficient than MANTIS. In the worst case, for a node at position $n = 8$ and a sensing task with the size of $l_s = 1ms$ TinyOS is 7.6% more energy-efficient.

If TinyOS is used, the length of the sensing task has little impact on the idle time measured. This is due to the low operating system overhead introduced by TinyOS. As previously mentioned, TinyOS spends nearly all available CPU time for application processing, not for operating system related tasks.

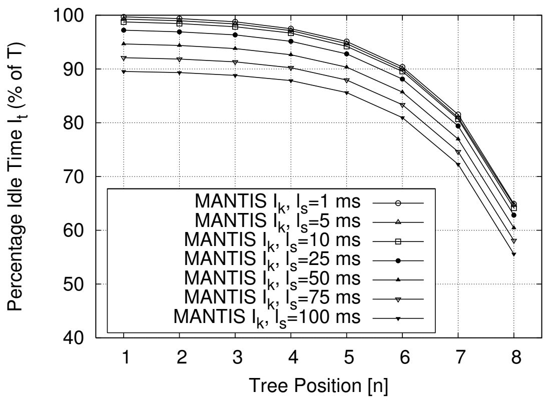
The theoretical maximum available idle time I_k^{max} and the measured idle time for TinyOS and MANTIS with a long sensing task of $l_s = 75ms$ is shown in Fig. 8 a). In Fig. 8 b), the resulting operating system overhead I_k^o is shown. As we can see, MANTIS has an increasing overhead with the increasing activity of the system. The TinyOS operating system overhead on the other hand is not very dependent on the load of the system.

Fig. 8 shows the average length of the idle periods I_p and the standard deviation of I_p for a short sensing task with the length $l_s = 1ms$. MANTIS has slightly shorter idle periods as it has a higher operating system overhead. Despite this difference, both operating systems achieve very similar sleeping periods. Also, the standard deviation in the idle period length is not very different. Thus, both operating systems can make use of available power modes in the same way. As shown, the thread-based MANTIS operating system does not fragment unnecessarily the available idle times. This effect however might be present in other thread-based operating system where a periodic context switch to a kernel-thread is performed.

E. Discussion and Findings

It can be confirmed that the multi-threaded MANTIS is not as power efficient as the event-based TinyOS operating system. However, the difference between both operating systems under specific conditions is not very big. If the system is not loaded (leaf node with $n = 1$ and a sensing task with the size of $l_s = 1ms$) a difference of only 0.09% in idle time is measured. Still, even if the system is in a heavy load situation (leaf node with $n = 8$ and a sensing task with the size of $l_s = 75ms$) only a 7.05% difference in the idle time is encountered. Thus, if an application is implemented where sensors are inactive for long periods and suddenly an event is detected that leads to an activity increase in the sensor field MANTIS is comparable to TinyOS in power efficiency. Additionally, as detailed in Section VI, MANTIS would be able to handle such bursty activity in a more deterministic way.

The experiments also show that the idle-times are not more fragmented in MANTIS than in TinyOS. Thus, the


a) MANTIS

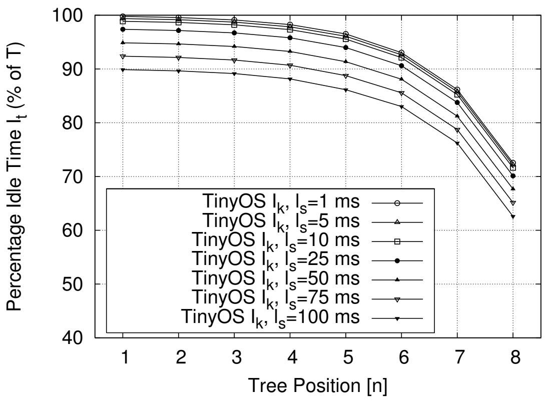
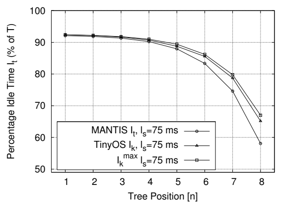
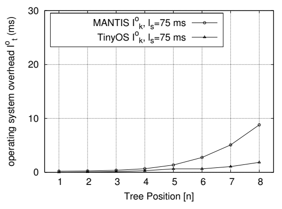
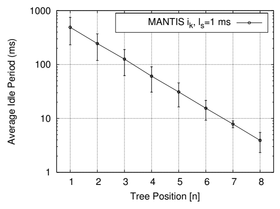

b) TinyOS

Fig. 7. Percentage idle time I_k for both operating systems.


a) I_k^{max} and I_k


b) I_k^o


c) MANTIS i_k

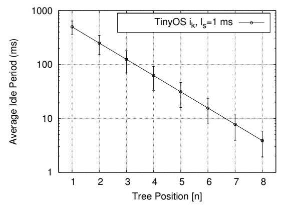

d) TinyOS i_k

Fig. 8. a) Percentage idle time I_k and the theoretical maximum idle time I_k^{max}, b) Operating system overhead, for both TinyOS and MANTIS with $l_s = 75ms$, c+d) Variation of Idle periods, i_k.

common argument that a multi-threaded operating system leads to high fragmentation of sleep times is proven wrong in the case of the MANTIS operating system. Both operating system types are able to exploit the same energy saving modes provided by the hardware.

Our findings regarding power efficiency of event-based and multi-threaded sensor network operating systems can be summarized as follows:

- The processing overhead resulting from a more complex threaded scheduler reduces the amount of time that a sensor node can sleep. The reduction depends heavily on the system activity.
- A multi-threaded scheduler has little impact on how fragmented a sleep schedule is. All energy saving modes provided by hardware can be accessed.

VIII. Conclusion

Most sensor network application scenarios used today are built using event-driven operating systems. The vast majority of these deployments use the event driven (and well established) operating system TinyOS. However, alternative operating system concepts such as the classical thread-based system exist. A well known example of a multi-threaded operating system is MANTIS. We believe a study, as presented in this paper, is needed to decide objectively which operating system type should be used for a particular application scenario. This decision is currently made on mainly subjective grounds for event driven systems and TinyOS. We believe this paper presents the first objective comparison between both main operating system concepts, taking the important performance parameters memory usage, event processing and energy usage into account. Based on the presented study we conclude that an event driven operating system is often, but certainly not always the best choice. Classic thread-based operating systems are of use for many sensor network application scenarios.

A. Results

Memory Usage: If memory efficiency is a primary goal of the target application then TinyOS would be the better operating system. In the experiment, the application scenario was compiled to a binary image 4kB less in TinyOS than MANTIS. While these memory requirements will have no impact on processors such as the Atmega 128, if the application is to be compiled for a more cost effective but less capable processor such as the PIC16 [22] then only TinyOS could be used. While both operating systems run a very light-weight scheduler, the TinyOS scheduler is smaller in code size as it does not provide a thread switching capability.

Event Processing: The processing time of packet forwarding tasks in the abstract application scenario depends mainly on the complexity of the sensing task, not on network activity. This dependency can be summarized as:

- If the sensing task is small (e.g. $l_s = 1ms$, $n = 1$), MANTIS has an 17.15% longer average packet forwarding task execution time than TinyOS. MANTIS and TinyOS have variation in the task execution time in the same order (Variation MANTIS $0\mu s$, Variation TinyOS $454\mu s$).

- If the sensing task is large (e.g. $l_s = 75ms$, $n = 1$) TinyOS has an 204% longer average packet forwarding task execution time than MANTIS. TinyOS has a much higher variation in the task execution time than TinyOS (Variation MANTIS $34\mu s$, Variation TinyOS $58ms$).

As a consequence, TinyOS is the preferred operating system in the case where no long sensing task has to be supported (Or in case the sensing task can be divided in small sub-tasks). In case long sensing tasks are executed, MANTIS would be the preferred system to achieve a deterministic and fast packet processing. Ultimately, the application requirements for the message forwarding performance of the network dictate which operating system can be used.

TinyOS has a high variation in the packet forwarding times as a pending packet task has to wait for a sensing task to finish. Especially when long sensing tasks are present, this delay is visible in the variation of the forwarding times.

Energy Usage: The energy usage of nodes in the abstract application scenario depends mainly on the network activity, not on the complexity of the sensing task. This dependency can be summarized as:

- If there is low network activity ($n = 1$, $l_S = 1ms$), TinyOS has an 0.09% longer idle time than MANTIS. Thus, TinyOS is more power efficient. The fragment size of idle periods is of the same order for both operating systems (Idle Time Length for TinyOS is $499ms$ and Idle time for MANTIS is $490ms$).
- If there is low network activity ($n = 8$, $l_s = 1ms$), TinyOS has an 7% longer idle time than MANTIS. Thus, TinyOS is more power efficient. The fragment size of idle periods is in the same order for both operating systems (Idle Time Length for TinyOS is $3.8ms$ and Idle Time for MANTIS is $3.9ms$).

As a consequence, TinyOS is the preferred in all operating conditions as it is more power efficient. The MANTIS operating system is not far behind in terms of power efficiency, especially if a low network activity is present. The more responsive MANTIS is not as power efficient as the necessary context switches reduce the available idle time. Especially when a large amount of network traffic is present, the context-switch overhead is prominent.

Summary: The difference in memory usage of the two operating systems is a (nearly) static parameter and does not depend on the particular application scenario supported. In most cases, this small difference would not be the deciding factor in choosing one of the operating systems. If a deterministic behavior regarding packet forwarding times is required, AND a complicated sensing task is carried out at the same time, MANTIS would be the better choice. This has to be paid with an additional energy consumption, but in some cases a deterministic network behavior would be preferred over a low energy consumption. If a TinyOS application requires a long sensing task, the poor packet processing times resulting from the non-preemptive scheduler, could lead to a congested network. This in turn would consume probably more energy retransmitting lost packets.

B. Application Scenarios

It was shown in Sections V, VI, VII that neither operating system would be optimal for all application scenarios as both operating systems target different performance objectives. In TinyOS for example, a key design metric is focused on compiling small application images, however the light-weight multi-threaded kernel in MANTIS is designed to provide predictable performance for more processing intensive applications.

To measure the operating system performance for a range of applications scenarios, the experiment evaluation used a range of sensing tasks sizes. These tasks times were chosen to model a spectrum of sensor network applications. Examples of such task times can be found in literature, for example encryption algorithms such as TinySec can take up to $1ms$ to encrypt a byte of data [17]. A structural monitoring application presented in [23] uses a wavelet decomposition algorithm that takes $12ms$ to compress vibration data readings, before transmitting data over the network. It can be concluded from our experiment that TinyOS would provide a more efficient scheduler for processing these applications. Such small task times achieve a better average response time and also more predictable response time in TinyOS due to the low processing overhead of the scheduler.

Image processing algorithms used in target tracking applications would be represented by the larger sensing tasks in the application scenario. Image processing algorithms are very processing intensive and difficult to realize on a sensor network. In [16] an object detection algorithm is used that can take up to $240ms$ on a 128x128 px image, $60.8ms$ on a 64 x 46 image and $16ms$ on a 32 x 32 pixel image. Scheduling a processing task of this magnitude in a non-preemptive scheduler such as in TinyOS can provide very ineffective schedules. We see from Section VI-D that high priority tasks such as a packet forwarding task can take a further $5ms$ to process when scheduled with a $75ms$ Sensing task. An obvious solution is to try and segment the sensing task, so that the high-priority task does not get delayed by the full duration of the sensing task. However in [16], the authors maintain that "Image processing operations are typically long-running and not suitable for sequential decomposition". In that paper, the authors realized their application on the TinyOS operating system with a dedicated ASP (application specific processor). The ASP processed all image algorithms to alleviate the CPU of all long sensing tasks and facilitate responsive processing of high-priority tasks. Adding an extra processor for all image processing sensor networks, considerably increases the cost of the sensor node platform. It might therefore have been more feasible to implement such algorithms with a multi-threaded operating system such as MANTIS.

C. Further Study

Both operating systems leave room for further study.

Our work described in [24] shows how preemptive scheduling capabilities can be added to TinyOS in order to tackle the performance problems described in Section VI. The established event driven processing concepts can be retained while adding preemption through context switching. Established TinyOS programming conventions can be used which ensures that existing application code can be re-used. Preemption features can be integrated seamlessly in existing TinyOS infrastructures.

Our work described in [25] shows how overheads due to context switching can be significantly reduced within the MANTIS operating system. Thus, the problems described Section VII can be diminished. The overhead can be reduced to such an extent that in usual sensor network application scenarios the modified MANTIS has a similar overall performance to TinyOS.

REFERENCES

[1] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," in *ACM SIGOPS Operating Systems Review*, vol. 34, pp. 93–104, December 2000.

[2] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, and R. Han, "MANTIS: System support for multimodal networks of in-situ sensors," in 2^{nd} *ACM International Workshop on Wireless Sensor Networks and Applications*, pp. 50–59, September 2003.

[3] A. Barroso, J. Benson, T. Murphy, U. Roedig, C. Sreenan, J. Barton, S. Bellis, B. O'Flynn, and K. Delaney, "Demo abstract: The DSYS25 sensor platform," in 2^{nd} *international conference on Embedded networked sensor systems*, pp. 314–314, November 2004.

[4] C. Duffy, U. Roedig, J. Herbert, and C. J. Sreenan, "An Experimental Comparison of Event Driven and Multi-Threaded Sensor Node Operating Systems," in 3^{rd} *IEEE International Workshop on Sensor Networks and Systems for Pervasive Computing (PERSENS2007), White Plains, USA*, Mar. 2007.

[5] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgenson, and R. Han., "MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms," *ACM kluwer Mobile Networks & Applications Journal, special Issue on Wireless Sensor Networks*, August 2005.

[6] C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in 3^{rd} *International Conference on Mobile Systems, Applications, and Services*, pp. 117–124, June 2005.

[7] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in 29^{th} *Annual IEEE International Conference on Local Computer Networks*, pp. 455–462, November 2004.

[8] A. Eswaran, A. Rowe, and R. Rajkumar, "nano-RK: An energy-aware resource-centric operating system for sensor networks," in 26^{th} *IEEE Real-Time Systems Symposium*, pp. 265–265, December 2005.

[9] S. Dulman, T. Hofmeijer, and P. Havinga., "AmbientRT - real time, data centric system software for wireless sensor networks," in 1^{st} *international Conference on Intelligent Sensors, Sensor Networks and Information Processing*, pp. 61–66, December 2004.

[10] A. Dunkels, O. Schmidt, and T. Voigt, "Using protothreads for sensor node programming," in *Workshop on Real-World Wireless Sensor Networks*, June 2005.

[11] E. Trumpler and R. Han., "A systematic framework for evolving TinyOS," in *IEEE Workshop on Embedded Networked Sensors*, pp. 61–65, May 2006.

[12] J. Regehr, A. Reid, K. Webb, M. Parker, and J. Lepreau, "Evolving real-time systems using hierarchical scheduling and concurrency analysis," in 24^{th} *IEEE Internation Real-Time Systems Symposium*, pp. 25–36, December 2003.

[13] L. Barello, "Avrx real time kernel." http://barello.net/avrx/, 2006.

[14] S. Li, R. Sutton, and J. Rabaey, "Low power operating systems for heterogeneous wireless communication systems," in *Workshop on Compilers and Operating Systems for Low Power*, pp. 1–16, September 2001.

[15] Atmel Corporation, *Atmega128 Datasheet*, rev n ed., March 2006.

[16] M. Rahimi, R. Baer, O. I. Iroezi, J. C. Garcia, J. Warrior, D. Estrin, and M. Srivastava., "Cyclops: In situ image sensing and interpretation in wireless sensor networks," in *In proc.* 3^{rd} *international conference on Embedded Networked Sensor Systems,*, pp. 192–204, November 2005.

[17] C. Karlof, N. Sastry, and D. Wagner, "TinySec: A link layer security architecture for wireless sensor networks," in 2^{nd} *ACM Conference on Embedded Networked Sensor Systems*, pp. 262–175, November 2004.

[18] Nordic Semiconductor, *Datasheet NRF2401*, rev 1.1 ed., June 2004.

[19] "The gnu project, http://www.gnu.org/," 2006.

[20] 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 Conference on Programming Language Design and Implementation*, pp. 1–11, June 2003.

[21] Microchip, *PIC16F877 Datasheet*. Microchip, f ed., October 2003.

[22] C. Lynch and F. O'Reilly, "PIC-based TinyOS implementation," in 2^{nd} *European Workshop on Wireless Sensor Networks*, pp. 378–385, 31^{st} Jan.-2^{nd} Feb. 2005.

[23] N. Xu, S. Rangwala, K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin, "A wireless sensor network for structural monitoring," in *ACM Conference on Embedded Networked Sensor Systems*, pp. 13–24, November 2004.

[24] C. Duffy, U. Roedig, J. Herbert, and C. J. Sreenan, "Adding Preemption to TinyOS," in *Proceedings of the The Fourth Workshop on Embedded Networked Sensors (EmNets2007), Cork, Ireland*, June 2007.

[25] C. Duffy, U. Roedig, J. Herbert, and C. J. Sreenan, "Improving the energy efficiency of the MANTIS kernel," in *Proceedings of the* 4^{th} *IEEE European Workshop on Wireless Sensor Networks (EWSN2007), Delft, Netherlands*, Jan. 2007.

Cormac Duffy received his B.Sc degree in Computer Science from the Dublin Institute of Technology, Ireland and M.Sc Degree from University College Cork. He had completed his Ph.D. at University College Cork on the topic of Lightweight Scheduling for Wireless Sensor Networks at the time of his sudden tragic death in August 2007.

Utz Roedig is a Lecturer in the Department of Communication Systems and the Department of Computing at Lancaster University. He pursues research in the area of wireless sensor networks. A specific aspect of this research is the data transport performance of wireless sensor networks. Utz received his Ph.D. in Computer Science at Darmstadt University of Technology in 2002.

John Herbert received a B.Sc. (Physics, Mathematics) and M.Sc. (Physics) from University College Cork, Ireland and a Ph.D. (Computer Science) from Cambridge University, England. He has held positions as Research Associate at Cambridge University, International Fellow at DEC Systems Research Centre, California and Computer Scientist at SRI International, Cambridge. At present he is a Senior Lecturer in Computer Science at University College Cork.

Cormac Sreenan joined University College Cork as a Professor in the Department of Computer Science in August 1999, and was Head of Department from 2000-2004. Prior to that he was a Principal Technical Staff Member with AT&T Labs Research (in Florham Park, NJ, USA), and also worked at Bell Labs Research (in Murray Hill, NJ, USA). He received his Ph.D. at University of Cambridge in 1992.