Improving the Energy Efficiency of the MANTIS Kernel

Cormac Duffy[1], Utz Roedig[2], John Herbert[1], and Cormac J. Sreenan[1]

[1] Computer Science Dept., University College Cork, Cork
[2] InfoLab21, Lancaster University, Lancaster

Abstract. Event-driven operating systems such as TinyOS are the preferred choice for wireless sensor networks. Alternative designs following a classical multi-threaded approach are also available. A popular implementation of such a multi-threaded sensor network operating system is MANTIS. The event-based TinyOS is more energy efficient than the multi-threaded MANTIS system. However, MANTIS is more capable than TinyOS of supporting time critical tasks as task preemption is supported. Thus, timeliness can be traded for energy efficiency by choosing the appropriate operating system. In this paper we present a MANTIS kernel modification that enables MANTIS to be as power-efficient as TinyOS. Results from an experimental analysis demonstrate that the modified MANTIS can be used to fit *both* sensor network design goals of energy efficiency and timeliness.

1 Introduction

Sensor nodes must be designed to be energy efficient in order to allow long periods of unattended network operation. However, energy efficiency is not the only design goal in a sensor network. For example, timely processing and reporting of sensing information is often required as well. This might be needed to guarantee a maximum delivery time of sensing information from a sensor, through a multi-hop network, to a base-station. To be able to give such assurances, network components with a deterministic behavior will be required. The operating system running on sensor nodes is one such component.

Event-based operating systems are considered to be the best choice for building energy efficient sensor networks as they require little memory and processing resources. Hence, the event-based TinyOS [1] is currently the preferred operating system for sensor networks. Event-based operating systems are not very useful in situations where tasks have strict processing deadlines. Tasks are processed sequentially, a prioritization of important tasks to meet processing deadlines is not possible. Multi-threaded operating systems are more suitable if such requirements must be fulfilled. Thread preemption and context switching enables such systems to prioritize tasks and meet deadlines. The MANTIS [2] operating system is the first multi-threaded operating system designed specifically for wireless sensor networks. Unfortunately, MANTIS has a relatively high processing

K. Langendoen and T. Voigt (Eds.): EWSN 2007, LNCS 4373, pp. 261–276, 2007.

overhead for thread management. This processing overhead is directly related to reduced energy efficiency because of the relative increase in CPU activity.

This creates the dilemma that both design goals - energy efficiency and timeliness - can only currently be optimized independently. One is forced to choose which goal is of higher importance in the considered application scenario. Therefore, it would be good if the dilemma could be resolved by either making TinyOS more responsive or MANTIS more energy efficient. In this paper the later problem is solved: We present a MANTIS kernel modification to increase power efficiency. As the results show, MANTIS can be modified to be as power-efficient as TinyOS without impacting vital kernel functionality. Thus, the modified MANTIS can be used to solve both important sensor network design goals.

The next Section of the paper presents related work. Section 3 presets preliminary research comparing TinyOS and MANTIS regarding event processing capabilities and energy consumption. This comparison motivates the modifications of the MANTIS kernel for better energy efficiency. Section 4 explains in detail the MANTIS kernel. Section 5 presents and explains the MANTIS kernel modifications. Section 6 shows an evaluation of the modified kernel. Section 7 concludes the paper.

2 Related Work

Problems arise when a sensor network applications require to be energy efficient and have to provide timely processing capabilities at the same time.

One example of an operating system that tries to bridge the gap is Contiki [3]. Contiki is an event-based sensor network operating system that includes a threaded library that can be optionally compiled to facilitate multi-threaded applications. Thus multi-threaded capabilities can be selectively designated to specific processes, without the processing and memory overhead in all parts of the system.

A similar approach can be seen in [4,5]. In both works the TinyOS operating system is encapsulated in a multi-threaded kernel. The operating system is then scheduled as a thread such that it can be preempted by complex threads if required. Thus TinyOS still achieves preemption without sacrificing the lightweight scheduling characteristics. In summary, the research focus of [3,4,5] is to minimize the processing overhead of a multi-threaded system, by isolating only the processes that require multi-threaded capabilities. However no effort is made to reduce the overhead of the multi-threaded processes.

In [6] a programming concept called "proto-threads" is described 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. Proto-threads are, however, merely a thread abstraction. They do not provide thread preemption, thus complex processes cannot easily be multiplexed with high priority tasks without introducing blocking.

The research listed above tries to compromise between power-efficient event-based schedulers and multi-threaded schedulers. The work presented in this paper focus on the reduction of processing overheads in multi threaded sensor network operating systems.

3 Preliminary Research

The preliminary research investigates the differences of the multi-threaded MANTIS [2] and the event-based TinyOS [1] operating systems. More details on the preliminary research can be found in [7]. The experimental methodology is reused for the evaluation of the optimized MANTIS presented in Section 6.

3.1 Evaluation Goals

It is generally assumed that an event-driven operating system is very suitable for sensor networks because few resources are needed, resulting in an energy-efficient system. However, the exact figures are unknown and therefore quantified in this preliminary research. On the other hand it is claimed that a multi-threaded operating system has good event processing capabilities in terms of meeting processing deadlines. Again, an in-depth analysis is currently missing and is therefore conducted. For comparison purposes, the event-based system *TinyOS* and the multi-threaded system *MANTIS* executing the same sensor network applications on the [8] are investigated.

The following parameters - while the sensor node is executing a generic application - are evaluated:

1. *Event Processing*: The average task execution time E_t of a particular reoccurring sensor task is measured. Average task execution time and its variance are a measure for the event handling capabilities of the system.
2. *Energy Consumption*: The percentage of experiment time I_t spent with an idle CPU is measured. CPU idle time can be used to suspend the CPU and thus relates directly to the energy efficiency of a system.

An application scenario for the evaluation has to be defined, as the parameters of interest are influenced significantly by the scenario. It was decided to use a scenario of a generic nature so that the results are applicable to a range of real-world applications.

3.2 Evaluation Setup

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 execute 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 $1ms$ [9]. 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 $100ms$ [10]. Other application examples documented in the literature are situated in between these values. Note that a long sensing task can be split-up into several sub-tasks but in practice this is not always possible. The experimental evaluation spans the task sizes described ($1ms$...$100ms$).

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 using TinyOS and MANTIS on the DSYS25[8] sensor platform for evaluation.

Topology. The sensor network is used to forward sensor data towards a single base-station in the network. It is assumed that a binary tree topology is formed in the network (see Fig. 1). Depending on the position n in the tree, a sensor node might process varying amounts of packets. Nodes closer to the root are more involved in packet forwarding as these nodes have to multiplex packet forwarding operations with their sensing operations. In the experiments, the behaviour of a single node at all possible positions n is emulated and measured by applying the sensing pattern and network traffic as described next.

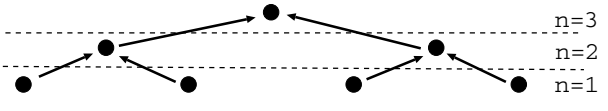


Fig. 1. Binary Tree

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 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 exists. 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 a good method to scale the processing performance requirements of a sensor network application. It is assumed that the duration (complexity) l_p of the packet-processing 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 [11].

3.3 Event Processing

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.

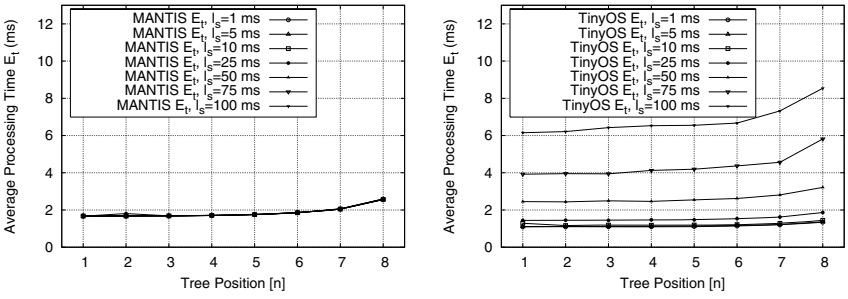
Task Execution Time. To characterize processing performance of the operating system, the average task execution time E_t of the packet forwarding task, is measured. During the experiment, J packet-processing times e_j are recorded. To do so, the task start time e_{start} and the task completion time e_{stop} are measured and the packet-processing time is recorded as $e = e_{stop} - e_{start}$. The average task execution time E_t is calculated at the end of the experiment as: $E_t = \sum e_j / J$. For each tree position n, the experiment is run until $J = 25000$ packet-processing events are recorded.

Results. In the experiment, the average task execution time E_t is determined for TinyOS and MANTIS supporting the abstract application scenario (see Fig. 2).

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 independent from the position n of the node in the tree. The average processing time increases slightly, under a heavy load. This is due to the fact that under heavy load packet forwarding tasks have to be queued (see Fig. 2 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 tasks also contribute somewhat to the average processing time (see Fig. 2 b)).

The variance in the packet-processing time E_t is also recorded but is not shown due to space restrictions. However, it has to be noted that this variance



a) MANTIS b) TinyOS

Fig. 2. Average packet-processing time E_t

is significantly smaller in MANTIS than in TinyOS (see [7] for details). Thus, MANTIS is better able to support scenarios which require predictable processing behaviour.

The thread prioritization capability of MANTIS is clearly visible in the experimental 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.

3.4 Energy Consumption

To evaluate power-efficiency, This study investigates the available idle time in which low-power operations can be scheduled. Thus the comparative effectiveness of specific power management policies can be guaged on the amount of potential low-power (idle) time available.

Idle time. 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 $i = i_{stop} - i_{start}$. 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: $I_t = (\sum i_k/T) \cdot 100$. Again, for each tree position n, the experiment is run until $J = 25000$ packet-processing events are recorded.

Results. In the first experiment, the percentage idle time I_t is determined for TinyOS and MANTIS supporting the abstract application scenario. (see Fig. 3).

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. 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

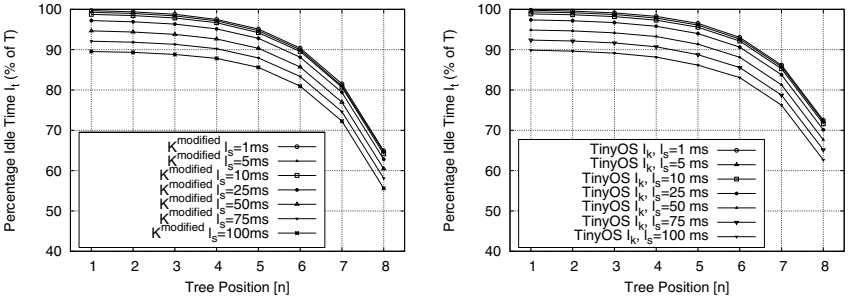


Fig. 3. Percentage idle time I_t for both operating systems

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.

3.5 Findings

The experimental results show that MANTIS has a much more predictive behavior executing the packet-processing task than TinyOS. More precise, the execution time in MANTIS has a low variation and is independent of other activity such as the sensing task. Thus, MANTIS would be preferable in situations that need deterministic and timely processing. However, the MANTIS system is not as power-efficient as TinyOS. Thus, TinyOS would seem preferable if energy consumption is deemed to be of primary importance. 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 0.1% in idle time is measured. However, if the system is under a heavy load (leaf node with $n = 8$ and a sensing task with the size of $l_s = 100ms$) a 6.9% difference in the idle time is encountered. The biggest difference is measured for $n = 8$ with a task size of $l_s = 1ms$ which results in a difference of 7.6%.

4 The MANTIS Kernel Architecture

The threaded MANTIS architecture implements thread-preemption, allowing the operating system to interrupt any active thread to immediately begin processing a thread of higher priority. As a result, the operating system can respond faster to critical events. In general, the system architecture follows the design principles of classical multi-threaded operating systems. However, to facilitate the necessary power management requirements, energy saving mechanisms are integrated in the thread scheduling. The processing states (e.g. sleeping, waiting) of all threads are monitored and used to decide which power saving modes of the CPU should be activated. Power saving is activated through a so-called *idle task* which is special purpose thread with the lowest possible thread priority, that is scheduled when all other threads are inactive.

Algorithm 1. Thread structure

```
 1: mos_thread_new(thread_A,128, PRIORITY_HIGH)

 2: thread_A
 3:   while(running)
 4:     ...
 5:     mos_semaphore_wait(A1)
 6:     ....

 7: int_A
 8:   ...
 9:   mos_semaphore_post(A1)
10:   ...
```

part A

```
1:dispatch_thread()
2:   PUSH_THREAD_STACK()
3:   CURRENT_THREAD = readyQ.getThread()
4:   CURRENT_THREAD.state=RUNNING
5:   POP_THREAD_STACK()
```

part B

4.1 Overview

Each task the operating system must support can be implemented as a separate MANTIS thread. A simplified view of this thread structure is shown in Alg. 1, part A. A new thread is initialized via the function *mos_thread_new* (line 1). Subsequently the thread processing, often implemented as an infinite loop, is started (line 3). Processing might be halted using the function *mos_semaphore_wait* when a thread has to wait for a resource to become available (line 5). An interrupt handler (line 7) using the function *mos_semaphore_post* (line 9) 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.

As an example, a thread might be used to process incoming packets from a transceiver chip. In this case, the *mos_semaphore_wait* is used to suspend the thread until a new packet arrives at the transceiver. If the transceiver receives a packet, an interrupt is executed and the thread is resumed to read the currently available packet and process it.

4.2 Scheduling

Thread scheduling is performed within the kernel function *dispatch_thread* shown in Alg. 1, part B. This function searches a data structure called *readyQ* for the highest prioritized thread and activates it. The *readyQ* is an array of linked lists containing pointers to the currently active threads. Each index of the array corresponds to a thread priority level.

When the *dispatch_thread* function is called, the current active thread is suspended calling *PUSH_THREAD_STACK* (line 2). Thus, the current CPU register information is saved to the heap memory allocated to the current thread. The highest priority thread is then selected from the *readyQ* (line 3) and its register values are restored by the *POP_THREAD_STACK* function (line 5). The thread can then resume processing at the exact point it was previously suspended.

Algorithm 2. Semaphore

```
1: mos_semaphore_wait(Semaphore s)
2:   s.val--
3:   if (s.val<0)
4:     s.addThread(CURRENT_THREAD)
5:     CURRENT_THREAD.state=BLOCKED
6:     update_sleep_counters()
7:     dispatch_thread()
```

part A

```
1: mos_semaphore_post(Semaphore s)
2:   s.val++
3:   if (s.getThread()!=NULL)
4:     s.getThread().state=RUNNING
5:     readyQ.addThread(s.getThread())
6:     update_sleep_counters()
7:     dispatch_thread()
```

part B

Algorithm 3. Timer Interrupt

```
1: t_slice_int()
2:   readyQ.addThread(CURRENT_THREAD)
3:   update_sleep_counters()
4:   dispatch_thread()
```

Before the *dispatch_ thread* function is called, the *readyQ* structure is updated. Threads that are currently sleeping or that are waiting on a semaphore are excluded from the *readyQ*. The scheduling through the *dispatch_ thread* function can be initiated by two different means: initiation within a semaphore operation or initiation through a time slice timer event.

Semaphore. A thread uses the function *mos_ semaphore_ wait* to coordinate access to a shared resource. If the resource is not ready, processing is suspended until the resource associated with the semaphore becomes available (Alg. 2, part A). If the resource is not immediately available (, line 3), the current thread is suspended and a context switch using the previously explained function *dispatch_ thread* is performed (line 7). Before the context switch is performed, the function *update_ sleep_ counters* is executed (line 6). This function is used to check if currently sleeping threads have to wake up and join the *readyQ* structure. In the MANTIS operating system, the user has the ability to make a thread sleep for a period of time. Thus, suspended threads either wait on a semaphore or they sleep. Pointers to the sleeping threads are stored in a sorted list, the *sleepQ*. Sleeping threads are sorted according to their wakeup time, such that the earliest thread to wake-up will be at the head of the queue. The function *update_ sleep_ counters* updates the wakeup times and if threads in the *sleepQ* are due, they are moved to the *readyQ*. Within an interrupt routine, *mos_ semaphore_ post* is called to inform a waiting thread that a resource is now available for processing (Alg. 2, part B). If a thread is waiting for the resource (line 3), the thread is activated and added to the *readyQ* structure. Thereafter, the *update_ sleep_ counters* function is called to check if sleeping threads have to be activated as well. Finally, the thread waiting for the semaphore (or a higher prioritized thread that was moved from the *sleepQ*) is activated using *dispatch_ thread*.

Time Slice Timer. A timer is set to create an interrupt every $20ms$ (Alg. 3). This interrupt serves two purposes. First, the interrupt acts as a time slice for

Algorithm 4. Modified semaphore functions

```
1: mos_semaphore_wait(Semaphore s)
2:   s.val--
3:   if (s.val<0)
4:     if(readyQ.getThread!=NULL)
5:       if(CURRENT_THREAD.state==BLOCKED_RUNNING)
6:         CURRENT_THREAD.state==BLOCKED
7:       s.addThread(CURRENT_THREAD)
8:       #ifdef MANTIS_SLEEP
9:         update_sleep_counters()
10:      dispatch_thread()
11:    else
12:      CURRENT_THREAD.state=BLOCKED_RUNNING
13:      do_power_management()
```

part A

```
1: mos_semaphore_post(Semaphore s)
2:   s.val++
3:   if (s.thread.state==BLOCKED)
4:     s.getThread().state=RUNNING
5:     readyQ.addThread(s.getThread())
6:       #ifdef MANTIS_SLEEP
7:         update_sleep_counters()
8:     dispatch_thread()
9:   else
10:    s.getThread().state=RUNNING
```

part B

the Round Robin scheduler, in which lengthly tasks are interrupted to give other equal priority threads a processing time-slice, thus preventing process starvation. Second, the periodic interrupts are used to check if threads in the *sleepQ* have to wake-up. The *update_ sleep_ counters* function is called from the timer interrupt to reactivate and reschedule sleeping threads. Obviously, threads sent to sleep using this mechanism do not expect to sleep with a period less than the periodic interrupt, $20ms$. Finally, *dispatch_ thread* is called to perform the context switch to the new thread. In many application cases, the new thread will be the same as the old thread.

4.3 Power Management

In MANTIS, thread state information is used to determine the level of power management to be initiated. Sensor network processors have a number of different low-power modes, providing a range of energy conserving states, varying in power-conserving performance and wake-up responsiveness.

Thread state information is used in MANTIS to determine if a thread requires a responsive wake-up, or if more relaxed wake-up times can be accepted. If the thread is *BLOCKED* (Alg. 2, part A:line 3), it is assumed that fast wake-up times are required and an idle power mode with fast wake-up response time is chosen. If all threads reside in a SLEEPING state, then the thread sleep counters are used to determine the next wakeup period. A timer is set to wakeup the processor in time for the next thread event. Thus the processor can be put into a deep sleep power mode and wakeup early enough to compensate for the slow processor wake-up period.

Power management in the MANTIS kernel is implemented as a separate thread, the idle thread. The idle thread is assigned the lowest priority and is always in a ready state. Thus, if no threads are ready to be processed the idle thread by default will be the next thread to be activated and the processor will be transitioned into a low power state determined as previously explained.

5 MANTIS Kernel Modifications

As shown in the preliminary research, MANTIS has the capability of task preemption and thus critical high priority tasks can be executed deterministically. However, the power consumption of a node running MANTIS is considerably higher than the power consumption of a node running TinyOS. The high energy consumption of the MANTIS operating system is caused by the processing overhead for thread management. This relatively high overhead is mainly caused by the (i) idle thread, the (ii) time slicing and the inefficient use of the (iii) kernel queuing structures.

5.1 Idle Thread

As previously explained, power management is implemented in MANTIS within the idle thread. If no other thread is currently active, the idle thread is dispatched which subsequently initiates the appropriate power-saving state. This method of power management is elegant as all power management code is contained in a thread but it is also highly inefficient.

When all threads are inactive (*SLEEPING* or *BLOCKED*), a context switch to the idle-thread is performed. Thereafter, as soon as one thread resumes activity, another context switch is required. The new active thread might even be the same thread that was active before the idle thread was called. Thus, for each sleep activity two context switches have to be performed which are in most cases not necessary on a typical sensor node running a single application.

To reduce the problem, the idle thread concept can be abandoned and threads initiate a sleep state directly. Thus, the kernel thread handling overhead can be greatly reduced, especially in scenarios where the same thread has to be activated after a sleep phase, avoiding context switching.

In the modified MANTIS kernel, the power-management procedure that was implemented as a thread is now implemented as a separate function that is invoked directly by the kernel when no more threads are available to process. The optimization requires a modification of the idle loop function. In the original MANTIS kernel the idle loop is initially invoked by the *kernel_ init* function to execute for the duration of operating system operation as a separate thread. In the modified MANTIS kernel, the idle loop is no longer encapsulated as a thread, but instead directly invoked from the kernel blocking procedures, i.e. *mos_ semaphore_ sleep* and *mos_ semaphore_ wait* (see Alg. 4). A new thread state is added to the kernel, the *BLOCKED_ RUNNING* state is used to signify if a thread can be reactivated after power management without a thread switch. A thread is first transitioned to this state when waiting for a semaphore while no other thread is active (Alg. 4, part A:line 12). Thereafter, the power management function is involved (line 13). If the processor is later reactivated and a resource is then ready, the *mos_ semaphore_ post* function will be called and the condition at Alg. 4 part B:line 3 will be used to determine if the blocked thread was already running before the power-management was invoked. If this is the case, all thread registers values still reside in the processor registers and a context switch is not

necessary. Instead, the thread state is changed to *RUNNING*, and the thread resumes processing.

5.2 Time Slice Timer

As mentioned, MANTIS creates a time slice interrupt every $20ms$ to alternately process threads of equal priority and addtionally update the *sleepQ*.

The periodic execution of the interrupt routine, and especially the necessary updates to determine which threads from the *sleepQ* have to be woken, represents a significant thread management overhead. Additionally, the *sleepQ* is also checked with each semaphore operation.

Round robin execution of equal priority threads is not really required in a sensor node. Either, one thread can wait for the other to finish execution or, if starvation is a concern, another priority level can be assigned to the thread. The sleep function using the *sleepQ* can be implemented alternatively using a timer interrupt combined with a semaphore. Therefore, the time slice timer can be removed from the MANTIS kernel without losing vital kernel functionality.

In the modified MANTIS kernel, the time slicing functionality and the associated sleep function using the *sleepQ* are removed. More specificly, this functionality is moved to a separate library that can be included in the kernel if needed. Applications can decide not to include the time slice timer and the associated sleep functionality in favor of more efficient processing. Such applications can therefor *not* invoke the *mos_thread_sleep* function to block a thread and must instead call a semaphore and a timer to block a thread for a predefined period of time. Equally prioritized threads in such applications execute sequentially until completion instead of being processed in a round-robin fashion.

To include the default MANTIS time slice timer, the user need only specify *#define MANTIS_SLEEP* in the application code. The *MANTIS_SLEEP* environment variable is used at part A:lines 8 and part B:6 in Alg. 4 to determine if the thread sleep functionality is required and the thread sleep counters must be updated with the *update_sleep_counters* function. Additionally, the time slice timer is set active.

5.3 The Kenel Queues

The MANTIS kernel maintains 3 types of link-list quing structures. The readyQ, sleepQ and semaphore queue are used to store threads in a READY, SLEEPING or BLOCKED state respectivly. A thread cannot reside in more than one queue at a time, and will therefore frequently switch between the queues as it changes state. As MANTIS normally handles a small number of threads (12 is the default number of threads supported [12]) simple data structures and ways of using them can be implemented. For example, all thread pointers can be kept in a simple array of pointers ordered by thread priority. Thread priority's and the number of threads normally do not change while an application is running and thus, the structure can be kept fairly static. Refrencing threads with static arrays requires far less processing that using a link-list.

In the modified MANTIS kernel all linked list structures are removed from kernel methods. The *readyQ* is changed from being an array of linked lists to a simple array of thread pointers. The thread pointers are kept permanently in the array while the threads exist. The thread pointers stored in this array are sorted regarding thread priority. Thus, addition and deletion of threads is costly but should not be common during a node's operation as threads are normally created at system startup. This change simplifies operations on the *readyQ* structure when semaphore functions are called. Threads need not switch between queues, a simple change of the thread state variable is all that is needed for a thread to switch state. The function *readyQ.getThread* (Alg. 4, part A:line 4) returns the first thread pointer in the *readyQ* array where the thread is in state *READY*.

If a thread is suspended and waits for a semaphore, the thread pointer is added to the semaphore structure. However, a copy of the thread pointer remains in the *readyQ*. The semaphore structure is also modified to point directly to a single thread rather than a link-list of multiple threads. Thus, the function *s.addThread* (Alg. 4, part B:line 5) is reduced to a simple pointer copy operation. The flexibility of using a semaphore to block multiple threads is obviously traded for efficiency.

6 Experimental Evaluation

The MANTIS kernel modifications are evaluated using exactly the same setup that was used in the preliminary research. Again, the average task execution time E_t and the percentage of experiment time I_t spent with an idle CPU are measured. According to the goals of the kernel modifications, the event processing capabilities of MANTIS should not be shortened and the energy consumption should be improved due to a reduction of processing overhead.

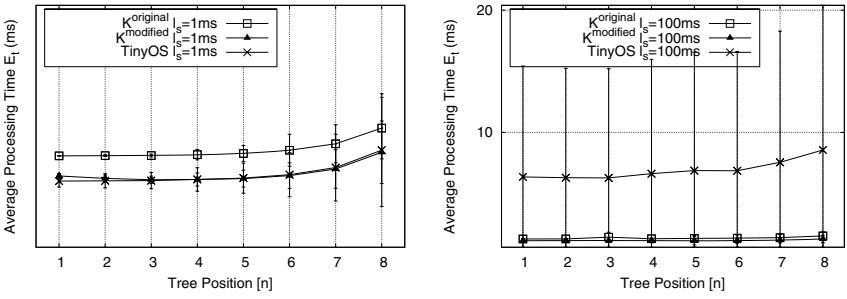
6.1 Event Processing

Fig. 4 shows the measured average packet-processing time E_t of the original and the modified MANTIS kernel for sensing tasks of two different sizes.

The results show that the average processing time of the packet forwarding task is reduced significantly. This decrease is due to the reduced processing overhead of the modified MANTIS kernel. The processing time is measured from the point the packet arrives to the time the packet is processed which includes possible context switching time (see Section 3.3). The pure packet-processing within the packet-processing thread accounts for $1ms$. Thus, the operating system can not execute the packet forwarding faster than $1ms$.

The trend in the packet-processing time is due to the fact that the packet-processing might sometimes preempt an active sensing task. Additionally, packet queuing effects become more dominant with increasing network load (an increasing n).

It can be deduced from the measurements that no significant difference in the variance of packet-processing timesbetween the orignial and modified MANTIS



a) $1ms$ sensing b) $100ms$ sensing

Fig. 4. Average packet-processing time E_t

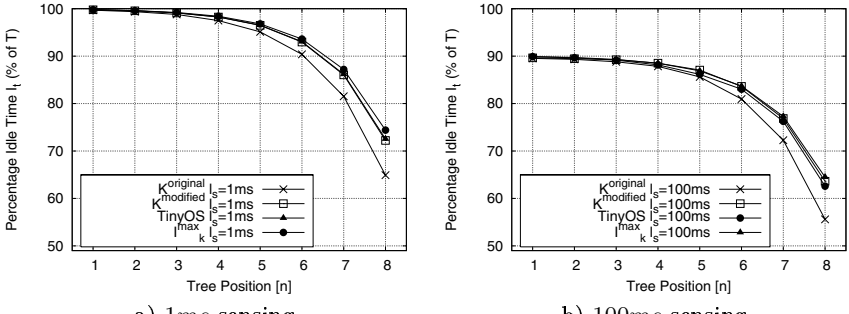
kernel(same magnitude of variation in the execution times). Additionally the processing speed is increased as the number of kernel overheads are reduced.

6.2 Energy Consumption

The percentage idle time is compared with the theoretical maximal possible percentage idle time, I_k^{max}. I_k^{max} is calculated by taking only the application processing of the abstract application scenario into account (see.Section 3.2). Thus, I_k^{max} represents the percentage running time that the processor would be idle using an ideal operating system which would have no operating system processing overhead. I_k^{max} depends on the task durations l_s and l_p of sensing and packet forwarding task respectively, 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):

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

Fig. 5 shows the measured average idle time I_t of the original and the modified MANTIS kernel for sensing tasks of two different sizes. Additionally, the maximum possible idle time I_k^{max} is shown in the graph.



a) $1ms$ sensing b) $100ms$ sensing

Fig. 5. Percentage idle time I_t

The results show that the available idle time is now very close to the theoretical maximum. The difference is especially visible under high network load (high n). The modified MANTIS kernel reduces overheads in context switches which is valuable in cases of a high system load.

Compared with the original MANTIS, the kernel modifications improve the idle time (by 8% for $n = 8$ with $l_s = 100ms$). Compared with the TinyOS operating system, the optimized MANTIS is now even outperforming TinyOS in some cases. For example for $l_s = 100ms$, $n = 8$, the modified MANTIS is 1% better than TinyOS. If $l_s = 1ms$, $n = 8$, the modified MANTIS is 0.3% worse than TinyOS under a heavy load.

7 Conclusion

As it is shown in the paper, it is possible to make a multi-threaded sensor network operating system as power-efficient as an event-based system. Thus, the commonly accepted fact that multi threaded systems are not useful for sensor networks due to their heigh energy consumption is invalid. Especially in scenarios that require timely event processing, multi threaded systems can be considered a useful option.

The MANTIS kernel modifications reduce the processing overhead needed for thread management dramatically. This overhead is reduced to such an extent that in usual sensor network application scenarios MANTIS has a similar overall performance to TinyOS. As kernel overhead is directly related to energy efficiency, the energy consumption of a MANTIS node is now similar to that of a TinyOS node. After the kernel modifications, MANTIS is 1% more energy efficient than TinyOS (in case of heavy load with $n = 8$, $l_s = 100ms$). With the original MANTIS kernel, TinyOS is 6.9% better than MANTIS (in case of heavy load with $n = 8$, $l_s = 100ms$).

We conclude that multi threaded systems can be used in sensor networks if designed carefully.

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

5. 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.
6. A. Dunkels, O. Schmidt, and T. Voigt, "Using protothreads for sensor node programming," in *Workshop on Real-World Wireless Sensor Networks*, June 2005.
7. C. Duffy, U. Roedig, J. Herbert, and C. J. Sreenan, "A performance analysis of TinyOS and MANTIS," tech. rep., University College Cork, November 2006.
8. 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.
9. Atmel Corporation, *Atmega128 Datasheet*, rev n ed., March 2006.
10. 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.
11. Nordic Semiconductor, *Datasheet NRF2401*, rev 1.1 ed., June 2004.
12. 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.