2014

A benchmarking tool for Wireless Sensor Network embedded operating systems

Mohamed Watfa
University of Wollongong, mwatfa@uow.edu.au

Mohamed Moubarak
American University of Beirut

Publication Details

Watfa, M. K. & Moubarak, M. 2014, 'A benchmarking tool for Wireless Sensor Network embedded operating systems', Journal of Networks, vol. 9, no. 8, pp. 1971-1984.

A Benchmarking Tool for Wireless Sensor Network Embedded Operating Systems

Mohamed K. Watfa
University of Wollongong in Dubai, UAE
Email: MohamedWatfa@UowDubai.ac.ae

Mohamed Moubarak
American University of Beirut, Beirut, Lebanon
Email: mam54@aub.edu.lb

***Abstract*—The emergence of the technology of Wireless Sensor Networks (WSNs) has lead to many changes in current and traditional computational techniques in order to adapt to their harsh and scarce requirements. A WSN consists of sensor nodes with wireless communication abilities that allow them to form a network. New system architectures have emerged to overcome sensor network limitations. Each architecture follows one of the two traditional design concepts, event-driven or thread-driven design. Although event-driven systems were assumed to generally perform better for embedded systems, tests have shown that event-driven systems tend to save more energy and space, while the thread-driven systems provide more concurrency and predictability, hence creating a tradeoff depending on the requirements of the application at hand. Performance analyzers are often used to accurately measure the performance of a certain system when such a tradeoff is evident. Performance analyzers can also locate deficiencies in a certain system for future improvements. The ever increasing complexity of applications executed by WSNs and the evolving nature of the underlying Embedded Operating Systems (EOSs) has led to the need for an accurate evaluation technique to guide practitioners in the field. This paper presents a novel approach towards providing a benchmarking and performance evaluation tool for comparing and analyzing the performance of WSN EOSs.**

***Index Terms*—Sensor Networks; Benchmarking; Performance Indicators; Operating System; Energy Limitations**

I. INTRODUCTION

Several embedded operating systems have been developed to manage the requirements of wireless sensor networks, based on different design philosophies and thus having different performances and tradeoffs. For that we intend to develop a performance measurement tool to accurately measure and evaluate the performance of those EOSs for comparative purposes. As a network, sensor nodes are expected to run a variety of sensing applications, reading in all types of data from acoustic to temperature values. To identify objects, a WSN will also need to do some pattern recognition, after which the sensors will diffuse the data on to a non-manageable network with low reliability. This requires the running of applications ranging from location aware algorithms to energy efficient routing. At the same time, a node may act as a router, forwarding data towards their destinations, or even responding to queries issued from base stations far away. Hence, a new specially designed OS is needed to operate all these states. The OS has to do so taking into consideration security, energy efficiency and high amounts of concurrency. This sounds like a blend of three types of OSs that exist today; personal computers, distributed systems and real-time systems. The required OS is also expected to run on a Memory Management Unit-less (MMU-less) hardware architecture, having a single 8-bit microcontroller running at 4MHz with 8 Kbytes of flash program memory and 512 bytes of system RAM [1]. Existing EOSs do not meet these requirements and hence the work on applicable OSs designed especially for WSNs has already begun.

Several EOSs for WSNs have been designed, implemented and in the process of enhancement. However, early before implementation, designers face an important decision to make. The designer of an EOS has to conform to one of two completely different design philosophies and build his system according to that philosophy. The two philosophies are called the event-driven and the thread-driven models. This decision is crucial in the sense that the behavior and performance of each model differs, and those will be reflected on the WSN since the EOS is the core of the system, and any protocol built on top of it will drag with it the characteristics of the design decision. Even at the application level, each model has its unique programming structure that programmers have to follow.

The rest of this paper is organized as follows. In Section 2, the necessary background about existing wireless sensor network operating Systems are outlined. In Section 3, the motivation behind our tool is discussed. Our statistical profiler is discussed in details in Section 4 and the experimental results are analyzed in Section 5. We conclude this paper in Section 6.

doi:10.4304/jnw.9.8.1971-1984

II. Sensor Networks Operating Systems

As a general assumption, event-driven OSs require fewer resources and less energy [6]. The energy efficiency of both TinyOS and MOS, which are WSN EOSs has already been investigated in the literature. To meet the tight constraints of WSNs, TinyOS adopted the event-driven approach as the concurrency model and is currently the standard OS for WSNs. TinyOS was designed to have a very small memory footprint, where the core OS could fit in less than 200 bytes of memory [1]. The event-driven choice in TinyOS was based on the fact that it cuts down on stack sizes since one process could run at a time. Also, the event-driven choice eliminates unnecessary context switches which are very energy inefficient. TinyOS is entirely made of a set of reusable system components and an energy efficient scheduler and hence has no kernel. Each component is made up of four parts, a set of commands, event handlers, a bundle of tasks and a fixed size frame for storage. The commands and events a component supports must be predefined to enhance modularity [14]. On the other hand MOS was the first thread-driven OS targeting the field of WSNs. The developers of Mantis believed that the threaded-driven model best suites the high concurrency needs of WSN applications. This design model eliminates the bounded buffer producer-consumer problem for example. The threaded design of MOS is useful as tasks for networked sensors become increasingly complex. Some nodes in WSN, for example, have to perform time consuming security encryption algorithms. In a system that allows only short tasks to run atomically, other time sensitive tasks may not be executed. MOS provides a unique characteristic compared to event-driven EOSs that is real-time operation. Real-time operation allows time sensitive tasks to execute within their assigned deadlines and thus is more predictable. Thread-driven systems are thought to have a memory footprint that is large enough to render them useless in the field of WSNs; however the developers of MOS were able to shrink a classic thread-driven OS into one that fits into 500 bytes of RAM [3]. So MOS' architecture is a traditional layered UNIX architecture. Like TinyOS, SOS (another WSN EOS) consists of components; however these components or modules are dynamically reconfigurable [2]. To achieve such reconfiguration, SOS consists of a statically compiled kernel, and a set of dynamically loaded modules. Another WSN EOS is RETOS. It was designed with four objectives in mind: provide a thread-driven interface, safe from erroneous applications, dynamic reconfiguration and network abstraction. A powerful set of characteristics for constraint networked sensors. What is unique about RETOS is the optimization techniques used to cut down on energy consumption and special resources. RETOS developers intend to make the technology of WSNs more popular by providing an easy programming model, thus the thread-driven model was their choice. Yet they also believed they have to optimize it to make it feasible. More EOSs for WSNs exist as well [14]. The following sections introduce the tradeoffs that may occur due to different EOS design philosophies.

A. Event-Driven Model

Event-driven systems are based on a very simple mechanism and are more popular in the field of networking. That is because the model complements the way networking devices work. An event-driven system consists of one or more event handlers. Handlers basically wait for an event to occur and hence they are implemented as infinite loops. An event could be the availability of data from a sensor, the arrival of a packet, or the expiration of a timer. Each event could have a designated handler waiting for it to occur. When an event occurs, the associated event handler either starts processing the event accordingly or adds the event to a buffer for later execution. Events are removed from the buffer in a FIFO manner. Task preemption in this model may occur if an event that has a higher priority occurs. The execution model is therefore rather sequential. Some well known examples of event driven OSs include:

- TinyOS: To meet the tight constraints of WSNs, TinyOS adopted the event-driven approach as the concurrency model and is currently the standard OS for WSNs. TinyOS was designed to have a very small memory stamp, where the core OS could fit in less than 200 bytes of memory [1]. TinyOS' event-driven choice was based on the fact that it cuts down on stack sizes since one process could run at a time. Another fact is that it eliminates unnecessary context switches which are infamous for their energy inefficiency. TinyOS is entirely made of a set of reusable system components and an energy efficient scheduler and hence has no kernel. Each component is made up of four parts, a set of commands, event handlers, a bundle of tasks and a fixed size frame for storage. The commands and events a component supports must be predefined to enhance modularity.
- SOS: SOS is another event-driven OS targeting WSNs. Like TinyOS, SOS consists of components; however these components or modules are dynamically reconfigurable [3]. To achieve such reconfiguration, SOS consists of a statically compiled kernel, and a set of dynamically loaded modules.

B. Thread-Driven Model

The thread-driven model is process based. Processes run preemptively on the CPU in a seemingly parallel manner. That is each process is given a quantum, which is an amount of CPU time. When the quantum ends, the process must be preempted and another process is run. Preemption in thread-driven systems occurs more than is strictly needed; however this CPU sharing provides the virtualization of several CPUs existing instead of one real CPU. The main part of a thread-driven model, or the heart of the system, is the kernel. The kernel provides all the system services such as resource allocation needed by the application level. The scheduler is the main controller of the system and it is built inside the kernel. It decides when to run a process and when to preempt it. Some well known examples of thread driven OSs include:

- Mantis OS (MOS): MOS is the first thread-driven OS targeting the field of WSNs. The developers of Mantis believed that the threaded-driven model best suites the

high concurrency needs of WSN applications. As mentioned in section II-C, this design model eliminates the bounded buffer producer-consumer problem. The threaded design of MOS is useful as tasks for networked sensors become increasingly complex. Some nodes in WSN for example have to perform time consuming security encryption algorithms. In a system that allows only short tasks, other time sensitive tasks may not be executed. MOS provides a unique characteristic compared to event-driven EOSs which is real-time operation. Real-time operation allows time sensitive tasks to execute within their assigned deadlines and thus is more predictable. Thread-driven systems are thought to have a memory footprint that is large enough to render them useless in the field of WSNs; however the developers of MOS were able to shrink a classic thread-driven OS into one that fits into 500 bytes of RAM [4]. So MOS' architecture is a traditional layered architecture.

- RETOS: RETOS is another thread-driven OS specially designed for WSNs. It was designed with four objectives in mind: provide developers with a thread-driven interface, safe from erroneous applications, dynamic reconfiguration and network abstraction. A powerful set of characteristics for constraint networked sensors. What is unique about RETOS is the optimization techniques used to cut down on energy consumption and space footprint. RETOS developers intend to make the technology of WSNs more popular by providing an easy programming model, thus the thread-driven model was the choice. Yet they also believed they have to optimize it to make it feasible. Operating systems supporting both the advantages of event-driven and thread-driven models of execution are highly desirable in WSNs. However, merging both models in one OS has led to merging the disadvantages as well. An example of such OS is:

- Contiki: Contiki is built around an event-driven kernel, moreover it provides optimal preemptive threading that can be applied to individual processes [8]. Contiki consists of a kernel, libraries, the program loader and a set of processes. It does not provide a hardware abstraction layer; instead it allows device drivers and applications to directly communicate with the hardware.

C. Event-Driven vs. Thread-Driven

Event-driven models are reputed by some researchers to provide more concurrency than thread-driven models do. However, other practitioners believe the opposite. To have a good idea about the tradeoffs of each design, we analyze the advantages and disadvantages of each model. It is generally assumed that an event-driven OS requires fewer resources and less energy [9]. The energy efficiency of both TinyOS and MOS' schedulers has already been investigated in the literature. The experiments were based on an abstract application that simulates network traffic. The application was run on each OS and the percentage idle time was calculated. The idle time is determined when there are no tasks to perform in both OSs. The more the OS spends in idle time, the more energy it saves. The application varies the amount of traffic and thus varies the position of the node on the virtual routing tree. The closer to the root the more the traffic is, while a leaf node means less traffic. So by manipulating traffic, a node can be repositioned on the virtual routing tree. The application consists of two parts, the sensing task and the arrival rate of packets. High traffic simply means long sensing tasks with high arrival rate. The results are in Figure 1.

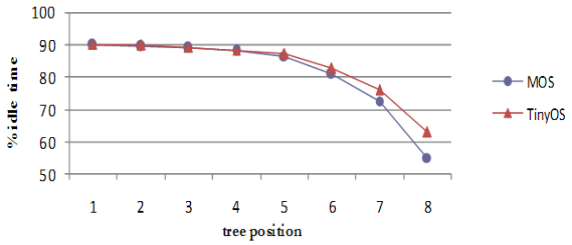


Figure 1. As traffic increases, MOS tends to spend more energy than TinyOS Due to the overhead of context switches

Figure 1 shows the result of the experiments when the sensing task is 100 ms long. As the number of incoming packets increases, TinyOS shows better energy consumption than MOS. This is because when the number of processes increases, the scheduler will do a context switch more often. Context switches consume more CPU cycles than most operations. However, when traffic is low, both OSs have similar performance. More experiments have shown that although MOS is less energy efficient, it is much more predictable than TinyOS in real-time operation.

D. Future of Sensor Network Operating Systems

Although several OSs exist for WSNs, yet most were developed as bases for future directions in WSN EOS design. Researchers are now interested in enhancing those EOSs for better energy consumption, space footprint and real-time operation, focusing on the operation of single nodes. From the design point of view, WSN OSs should have the characteristics of distributed systems which are still not evident in present WSN OSs. More research needs to be conducted on the feasibility of sharing resources among WSN OSs. Moreover, WSNs in the future will consist of thousands of nodes. Having so many different OS designs and execution models and different performances requires research on hybrid deployment to allow for the scalability of WSNs. Better yet, a global design for WSN OSs could be engineered. TinyOS for example is already noticed as the standard OS for WSNs and most research effort is done on top of it. To achieve a general design, more work should be done in eliminating the resource/accuracy tradeoff between different OSs such as the one we saw earlier where MOS provides more accuracy than TinyOS, which in turn provides better energy consumption than MOS. This could be done by optimizing preemption in thread-driven systems or by adding preemption to event-driven systems. Adding preemption to TinyOS is an ongoing research effort. Moreover, a more reliable comparison for WSN OSs is needed in order to pinpoint other tradeoffs and hence build a clearer picture of the intended general system. To do so, research on creating a performance analysis for WSN OSs is indeed of great interest. From

the programming point of view, current programming models are too low-level. For example, we saw how Contiki allows the programmer to directly manipulate the hardware without a hardware abstraction layer. Although this may decrease the number of levels in the hierarchy, but it forces the developers to think about hardware details. This is reflected by the huge effort put in order to create demos [12]. We need a programming model that eliminates the irritable details of hardware. Moreover, current programming models are also node-centric such as nesC for example. nesC focuses utterly on programming individual nodes. One area for research in this field is macro-programming. The main purpose is to develop a high-level language to implement aggregate programs for a WSN. One such work called TinyDB has already taken this step [12]. More programming models that target an entire system are needed. Other programming tools for WSN programming are also at a stage that requires more work to be done such as tools for debugging and programming interfaces (IDEs). Although research in the field of WSN EOSs is actively growing, practitioners in this field have very few choices in terms of OS and development tools. A practitioner thus has to understand the principles of each OS and its programming model to make a choice and present better results depending on his requirements. The main goal of presenting different OSs was not to decide which is superior to the other, but to show what conventional methods apply and what novel methods present as an alternative. These were the same goals the developers of the discussed OSs had in mind. This is because they were motivated by the resource challenges presented by WSNs. Now, after all the efforts done to find out what is feasible and what is not, we may say that the motivation for current research has partially shifted from challenging low-level constraints of the nodes to higher-level constraints of OSs. Our paper solution provides a flexible tool as a building block for all the aforementioned issues. The previous sections describe what a programmer has to go through to decide on the programming language issue. However in terms of performance, the average developer has no means of figuring out or predicting how his application would behave in real life deployment. As for performance engineers, the process itself is very hasty. Without a performance analysis tool, one needs to implement a specific experiment each time, going into the lowest details in order to pinpoint a bottleneck in the system if one exists. Such a process is time costly and may take huge of research efforts.

E. Summary of Observations

In our earlier work [27], we have shown how the value of preemption has a great impact on the design and implementation of operating systems. We introduced a simple and energy efficient preemption algorithm targeting embedded wireless sensor network operating systems. We implemented our algorithm on an embedded operating system and evaluated its performance. Our algorithm is general and portable in the sense that it can be applied on any preemptive platform. Moreover, we have showed a significant decrease in the number of context switches using our algorithm. Our algorithm also maintains the predictable nature of the preemptive system. We claimed at the beginning that optimizing a system is a tedious process that requires the involvement in low level details. In both our work [27] and in [5] a specific benchmark was implemented to measure the performance of the schedulers. The benchmark is specific to the extent that it is completely useless after the results are gathered. Moreover, it cannot be used on other operating systems. Another important aspect that we were emphasizing is that evaluating the performance of a system is tedious and requires a lot of research efforts. For example, finding the tradeoff between TinyOS and MOS required detailed knowledge about the systems from instruction level to the highest level of coding. After which a prediction had to be made on the source of the tradeoff. We experienced this after implementing our scheduling algorithm. We needed a way to evaluate the performance of the system before and after. And again we had to develop our specific benchmark application. The goal of this paper hence is to develop a portable tool that can be used to measure the performance of a system while running any application in a real life deployment as well.

III. Performance Analysis

The motivations and challenges each OS designer had in mind was the main reason behind the existing variety of WSN OSs. Other than realizing the constraints of WSNs and fitting a UNIX kernel in 500 bytes, the main motivation was and always is to achieve maximum performance. Moreover the large number of potential design and the relentlessly sprouting nature of workloads have resulted in performance evaluation becoming an irresistible task for WSN EOS. Two approaches exist to analyze the performance of a system. The first approach is called Performance modeling and is based on simulations. The second approach is called performance measurement and is based on the real nodes. The second approach could be offline or online. The offline approach involves attaching special hardware to the node, where the online approach is software based and involves special software on top of the real system. The latter is subdivided into two subcategories namely benchmarking and profiling.

A. Performance Modeling

Performance evaluation techniques have been evolving ever since they were first used in the mid 1940s [50], where throughput was analytically measured to determine the performance of scientific calculations. Today, the de facto standards for performance evaluation are simulation techniques. This approach uses a simulator that models the system under test, written using a high level language such as C and runs on top of different hardware. Such an approach could simulate every cycle of the processor under study. A trace produced by a profiler may be fed to the simulator which in turn produces the performance results. Hence the simulator models the system and the trace models the software or the benchmark that will be run to generate the results. Such simulations have a set of drawbacks. First, the simulation approach is time

consuming. Benchmarks usually consist of large programs that constitute of millions of instructions that have to be simulated. Second, this approach is mainly targeted at the performance of different CPU architectures and not OSs. This is because simulators depend on the OS of the machine running the simulation to allocate resources and the original OS itself is not simulated. Examples in the field of WSNs include XMOS and TOSSIM. Simulators in general fail to replicate real life execution. This is especially apparent in WSN simulators. WSN simulation based executions are by far different from their real life execution.

To understand why, let us discuss the execution model of TinyOS applications. The main reason why Simulations do not represent the real execution of WSN applications is because of the dependence of these networks on the external environment. A sample deployment of a WSN could consist of thousands of sensor nodes in a forest for monitoring irregular weather variations. In such a deployment, each node can sense data and send the information to the other end of the forest through wireless communication providing a cheap method for monitoring large areas. Each node is considered an embedded device since it has a specific task to do, which is to sense a phenomena and communicate wirelessly. This implies that these devices are greatly influenced by the environment around them. WSN applications are concurrent, event-driven systems which frequently interact with their environments through interrupts and events that happen at arbitrary times. To analyze and test WSN applications, we need to be able to determine the impact of these interrupts on the code. If we want to follow the execution trace, or even the possible execution traces, in a traditional program, we would follow the function call sequence, since they are sequential programs. That is, when we see a function call in a program, we know for sure that the execution flow will proceed to that function and return to the caller where the execution will continue again from there. In TinyOS, however, we have what is called deferred tasks. Those are tasks that are posted for later execution. And hence a function call does not mean that the execution flow immediately proceeds to this function since it may be a deferred function call. Although this behavior could be simulated, what makes the simulation unrealistic is the effect of external events. The primary job of sensor nodes is to react to external stimuli. Upon the existence of such stimuli, the nodes sensing this effect will react accordingly. Hence, the environment controls the execution of WSN applications. In simulations, the environment does not exist. The developer manually triggers any stimuli. Real life scenarios however are completely unpredictable and thus the execution model of WSN applications is random as well. This characteristic does not only affect simulations, but as we will see later on, also affects benchmarking results where they have a larger impact. Although techniques that optimize the speed of simulations exist, simulation based performance evaluation is only applicable at early design stages before the system under study has been implemented and hence such technique is called performance modeling [47]. The other performance evaluation technique that is of importance to us is called performance measurement and is discussed next.

B. Performance Measurement

This technique assumes and uses an existing prototype and not one that is still being designed. There are several techniques to understand the existing system's performance using the performance measurement approach, three of which are described here: off-chip monitoring, on-chip monitoring and software monitoring [47]. The off-chip monitoring approach is done by attaching separate hardware modules to collect performance characteristics and numbers. Such hardware modules could be logical analyzers or hardware-profilers that interrupt the CPU and gather information at each interrupt. This type of data collection can significantly slow down the system under test. An example of such tool is JTAG, a widely used debugging tool for sensor node applications. A less common approach in the embedded field is on-chip monitoring. This approach relies on the capabilities of the underlying CPU in the sense that it uses special CPU counters to gather system data. This approach assumes that the system under study provides such counters. State of the art CPU vendors publish documents on the performance counters provided by their CPUs. Such counters do not require the source code for the benchmarks or workloads to be available. Software monitoring on the other hand uses interrupts or traps at the OS level to capture the desired data at the instance of the trap. This approach is independent of the application running. The system can still monitor applications by taking snapshots every interval. For large systems, such approach was more popular before the on-chip approach saw light. However, this approach best suites WSN systems as we will see later on. Another form of software monitoring or profiling is the benchmarking technique. The following sections focus on the software based performance analysis techniques, namely profiling and benchmarking, since they are the only feasible solutions for WSNs. Afterwards, we narrow our choice to one technique based on the differences. Many attempts have been done to provide some performance information for researchers about the designed OSs. Yet they are nowhere near being a decisive tool for practitioners and designers. The information is not presented in a comparable manner nor do the workloads used reflect the nature of real life applications. Let us before discuss the related work in the field which some researchers referred to as void [45].

C. Wireless Sensor Network Performance Analyzer

To illustrate the reason behind our choice of profiler, we will summarize the difference between the aforementioned techniques. All in all, there are four techniques; three of which have been used for wireless sensor networks. The first is the hardware approach. This approach is precise and automatic, however requires extra specialized hardware. Hardware needs to be attached to the node in order for this approach to be used. This means

the process is not scalable and the parameters that can be monitored are fixed. The second approach is based on simulators. As opposed to the hardware approach, this approach is more scalable. Simulators allow developers to simulate networks with a large numbers of nodes. However, developers cannot simulate the operating system itself and measure its performance. Simulators are used for network wide performance measurement. Even in that sense, simulators are a poor choice. This is because it is difficult to replicate the real life execution of a wireless node or network. Again, the reason is because wireless sensors by nature depend on the environment around them. They are listeners and react to natural stimuli which occur randomly and naturally. So their behavior cannot be predicted and simulated. This has been shown by researchers that try to generate control flow diagrams from the execution of wireless sensor nodes [44]. Due to the large impact of unpredictable natural stimuli, the execution paths of sensor node applications are unpredictable. The third approach we have mentioned is the software based benchmarking technique. The only advantage this technique introduces is that it could analyze a sensor network at runtime. However, specialized applications need to be running to collect and measure performance. The problem with the approach of benchmarking using specific applications is that wireless sensor networks run a variety of applications and a benchmark suite needs to include all this variety. To solve all these problems we implemented the first statistical profiler for wireless sensor nodes. Using a profiler as a tool integrated in the operating system, a wireless sensor network can be monitored at runtime. This means that results are based on real life scenarios. Moreover, the profiler is a piece of software that could run on all the nodes and hence it is scalable. The most important aspect is that as opposed to benchmarks, any application running could be monitored and not specialized suites need to be designed. Using such a tool, huge research efforts such as that in [5] [27], could be significantly reduced. The goal of this paper is hence to apply the state of the art performance measurement techniques to create a performance measurement tool to evaluate the performance of different EOSs targeted for WSNs. Specifically; we implemented a statistical profiler to measure the performance of WSN EOSs. One factor about WSNs that cannot be neglected is the harsh conditions or environments that they will operate under in real life. This work will be the first to collectively evaluate and standardize the performance of WSN OSs. The contribution of this work is twofold in the sense that it is seen from the point of view of both the system designer and the application programmer. From the designer's perspective he will be able to:

- Tune systems that have been built. According to the performance of the system, drawbacks in design can be pinpointed and tuned accordingly such as energy performance and speed. Since application performance depends on the underlying OS [50], the application's performance will be automatically tuned as well.
- Validate performance models (simulators) that were built and come up with optimization techniques. Simulation models and evaluations done prior to prototyping may be validated by the more accurate results of hardware monitoring.
- Validate analytical evaluation and measure analytical predictability. Again the performance results could be used to validate the analytical results as well as determine the margin of errors induced by mathematical simplifications.
- Use a more appropriate design philosophy for future systems. Performance numbers largely reflect the flaws in design and will also be able to determine the superiority among different design strategies.
- Understand the bottlenecks of the designed system. A rich benchmark will be able to locate flaw and not only indicate them.
- Understand the relation between the application and the system. The designer will be able to answer whether the performance of the system is application dependant or not.
- Determine not only flaws, but also the advantages of the system by understanding how the system exploits different applications.
- Evaluate the performance of applications that spend most of their time in the kernel level which is a feature being ignored my most benchmarks [50].
- Evaluate the performance of the three levels, hardware, OS and application.

From the application's programmer point of view:

- For the programmer to pick the OS that bests suites his application, the task seems tedious. The developer has to first identify the different OSs that exist. Then has to research the design philosophy behind each (event-driven or thread-driven). After determining the design philosophy used, he has to research which kind of application that design bests suits where he will find no satisfying answer of the everlasting debate. The proposed benchmark technique gives a direct answer for such programmer by introducing a single value. Moreover, sub values discussed later in the methodology can give a more specific answer directly related to the application in mind.
- A practitioner may save himself the hassle of determining which OS better suites his application and picks the design he is used to or the design with the more user-friendly language. This choice is completely vague and appears as the best choice for a practitioner in the field. The proposed benchmark will provide a much more accurate path for this type of problem.

IV. Wireless Sensor Network Statistical Profiler

In this section, we will go through building the statistical profiler, step by step. Figure 2 below illustrates all the components of our WSN performance analyzer. The first component is the timer wrapper. The timer wrapper is a driver that we implemented and will be used

by the WSN Analyzer as a frequency generator. The WSN Analyzer will collect information from the scheduler, log the information and send it to the base station for offline analysis and plotting. The remaining sections describe these steps in detail.

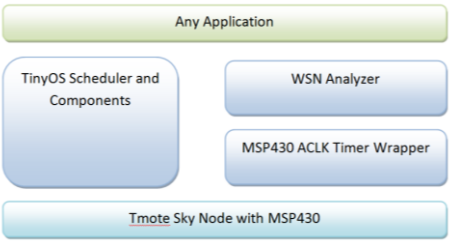


Figure 2. Structure of TinyOS and WSN analyzer

The core component for the statistical performance analyzer is the timer since the statistical analyzer requires a frequency supplier. It is better to use a timer independent of the system timer. This section describes the hardware platform we are using and the timers on that platform.

- Tmote Sky MSP430: The Tmote Sky features an ultra low power Texas Instruments MSP430 F1611 microcontroller. This 16-bit RISC processor performs extremely low power current consumption permitting a lifetime of years on a single pair of AA batteries. The MSP430 has an internal DCO that may operate up to 8MHz. The DCO can be turned off for low power consumption, where the MSP430 operates off an external 32 KHz oscillator. This is good news since we have a clock (32 KHz) independent from the system clock (8MHz) which is an ideal case for our tool. In the following section we briefly discuss how we will use the watchdog timer and the 32 KHz frequency source to create an interval timer as a core interrupt generator for our profiler.

- MSP430 Watchdog Timer: The Watchdog timer on the MSP430 platform is a 16-bit timer that can be used either as a watchdog or as an interval timer. A watchdog timer is usually used to restart the system if software problems occur. If the selected time interval expires, a reset is automatically generated by the watchdog timer. This means the some component has to reset the watchdog timer as an indicator that it is running correctly. Most importantly, the watchdog timer can be alternatively used to generate interrupts at a certain rate. Since this timer is also independent of the system timer, we use it as an interval timer for our tool. To do this however we need to implement a wrapper for the watchdog timer to make use of it as an interval timer.

The Watchdog timer has a register that contains control bits. The value of these bits determines the mode (watchdog/interval), the frequency and several other variables. This register is referred to as the WDTCTL register [30]. The clock source can be set using the WDTSSEL bit on the WDTCTL register. The sources are the SMCLK and the ACLK or the system clock and the auxiliary clock respectively. To set the timer into interval mode, the WSTTMSEL bit has to be set to 1. Unlike the watchdog timer, in interval mode, the node will not be reset when the interval expires. For example, the instruction to set the timer in interval mode and set the frequency of interrupts to 1 second is

```
WDTPW|WDTTMSEL|WDTCNTCL|WDTSSEL
```

This line of code fills the WDTCTL with a 1 bit value in the above named slots, which in turn sets the timer to interval mode with 1 sec interrupt rate. The entire WDTCTL register looks as follows:

```
WDTPW|WDTHOLD|WDTNMIES|WDTNMI|WDTTMSEL|WDT
CNTCL|WDTSSEL|WDTISx
```

For more information about the WDTCTL register on the MSP430, please refer to [30]. This forms the core of our analyzer and all that remains is the event handler that captures the interrupts. In this event handler, the code which takes a snapshot from the system resides. Now that we have the frequency source, an interrupt is generated every interval. The frequency could be set to 0.25, 0.16 or 0.019 seconds. For our tests we used the 0.019 second interval. As mentioned before, the performance analyzer is made up of an interrupt generator, which generates interrupts at a certain frequency, and an event handler, which captures the interrupt and takes the required snapshot. This snapshot basically consists of the name of the current running process. The profiler also requires a data aggregation and dissemination system. We have discussed the first part in the previous section. In this section we describe the event handler, and the logging/dissemination is discussed in the following section. The event handler is implemented as a separate module and runs automatically.

In TinyOS for example, this was done by wiring up the profiler with the boot module, this way the profiler starts when the system boots, even if no application was running, monitoring just the OS. The event handler's job is to store the name of the process that is currently running. This is done by first getting the name and then writing it to flash memory. The first part is not possible at runtime since most WSN Oss (except for Contiki) do not store the name of a process in the scheduler's queue. In TinyOS for example each task is given a unique id. This id is mapped to a task name offline, at the base station when the information is collected from the mote(s). The name is determined from the app.c file generated by the nesC compiler. In that file, each function is associated with an id and a name. The following is an excerpt from the app.c file of a TinyOS program:

```
#120 "opt/tinyos-2.x/tos/chips/msp430/timer/Msp430TimerP.nc"
Static inline void
/*Msp430TimerC.Msp430TimerB*/Msp430TimerP$1$VectorTimerX1
$fired(void)…
```

Hence the function fired from the module Msp430Timer has id 120. The event handler's job hence is to get the id of the currently running task from the task data structure. This is as simple as getting the variable 'head' from the system's queue.

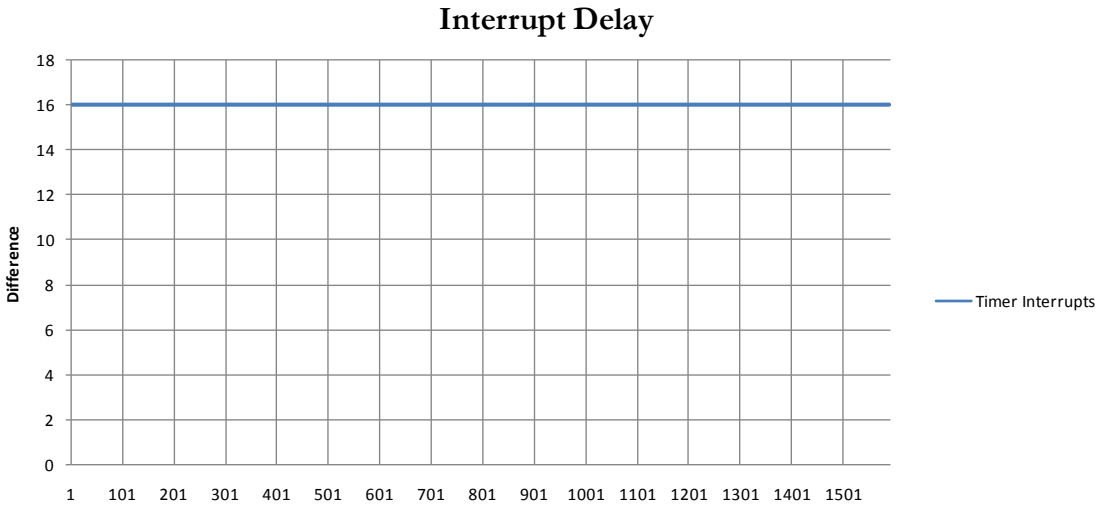


Figure 3. The interrupt generator has no delay due to its priority

A. Data Logging and Dissemination

TinyOS 2.1 provides several logging techniques by using the flash memory on the Tmote Sky nodes. One of these techniques is the circular logging implementation. The circular log implies that whenever the log is full, the oldest entry is replaced with the newest entry. This approach best suites our tool because the number of snapshots we will take will be large and will be cut down during the offline analyses hence will not hinder the results. This version of TinyOS also provides an implementation of the printf library. This library is used to send the data through the USB serial port to the base station, or to disseminate the data through the air. This requires a special implementation of the printf library which we have modified to use the air interface and to use the circular log approach. For our preliminary results, we have only tested using a single node sending data through the USB serial interface. Hence our preliminary experiments involved a single node, using the USB interface to send the data, which is being collected by the analyzer at a rate of 1.9 ms. The application monitored is a simple application that sends and receives sensor readings at a rate of 4Hz. This application uses no communication. The blink application sets a timer with an interval.

B. Experiments

Before plotting our preliminary results, we need to discuss our experiments. Our performance Analyzer was implemented on TinyOS, Mantis and Contiki. The first experiment was to test our frequency source. The timer implemented for our analyzer generates interrupts that are maskable. That is other non-maskable interrupts can make it wait. This may affect the results in the sense that some events cannot be monitored. This is a general problem that statistical profilers face when using maskable interrupt generators. This effect is called delay-effect. This effect can be measured by modifying the event handler as follows:

- Remove all the snapshot taking code.
- Insert a function call to get the current time.
- Log the current time and send it through the serial interface

By calculating the time differences between each interrupt, we can plot the results and see the variation in the intervals. A straight line will indicate correct operation, and a curved line will indicate a difference in the intervals. We expect to have no variation in the intervals because although the watchdog timer is maskable, it has a higher priority than other interrupts that could possibly mask its execution [30]. After we conducted this test, the results shown in Figure 3, show that indeed no other interrupts whether software or hardware were able to mask the timer's interrupt and cause a delay. This is due to the fact that the interrupt vector associated with the interval timer has a higher priority than most interrupts [30]. The interval used in this experiment was the 0.16 sec interval.

The remaining set of experiments were made to validate our assumptions. The first set of experiments measured bottlenecks. The second set of experiments was made to show how the tool could measure several different metrics. The following section describes these experiments in more detail.

V. PROFILING RESULTS

For our experiments, we ran the same application on TinyOS, MOS and Contiki. The application works as follows. At a rate of 4Hz, the application collects sensor readings from the onboard sensor, more specifically the temperature sensor. Once read, the value is stored in a packet and broadcasted immediately. While doing so the node is also listening for packets that might arrive. Upon receiving a packet, the listening nodes unpack the packet to get its payload which should contain temperature readings. Each value read is then displayed on the leds (least 3 bits). The application is not complicated hence tasks are very short. For this reason a large amount of time is expected to be in idle mode. The reason such an application was chosen is to compare the idle time between TinyOS and Mantis. This is to justify our claim

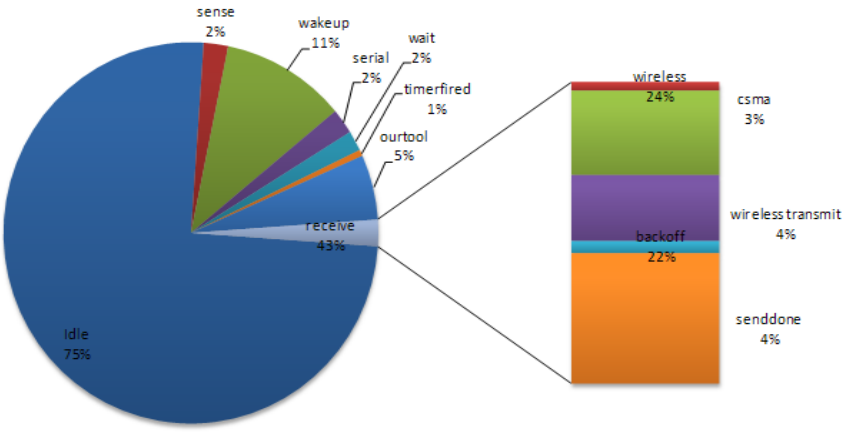
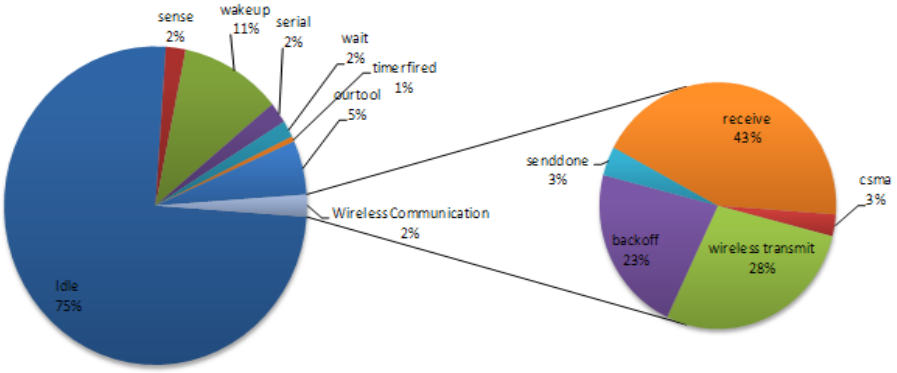
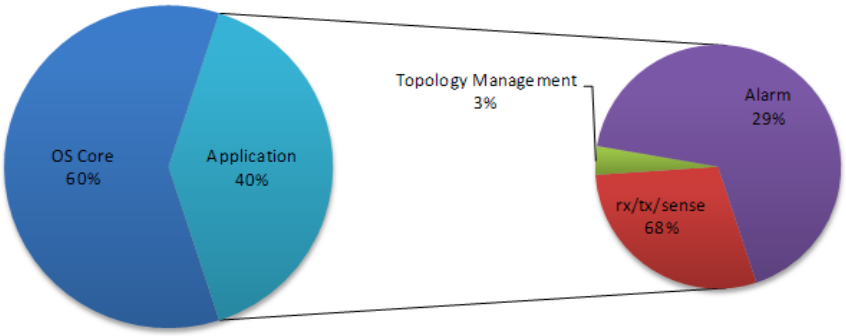


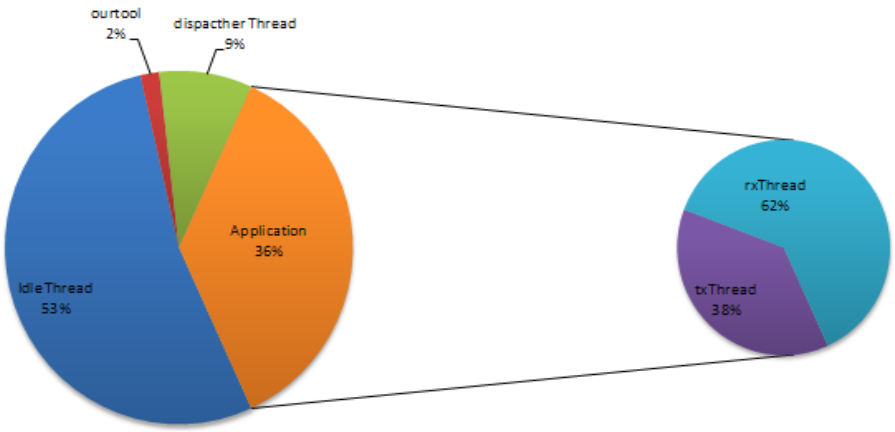
Figure 4. Profiling a simple TinyOS execution taking into account sleep times



(a)



(b)



(c)

Figure 5. (a) Profiling a simple TinyOS execution taking into account sleep times, (b) Profiling a simple Mantis execution taking into account sleep times, (c) Profiling a simple Contiki execution taking into account sleep times

that if Mantis spent less idle time, our tool will be able to pinpoint the reason why. In other words, our tool will tell us where the decrease in idle time is being complemented. Figure 4 below shows the results obtained from TinyOS. Figure 4. can be used to determine bottlenecks in the system which is one of the features of our tool, as expected most of the time was spent in idle mode. This also tells us something about our tool. That is although our tool is continuously running in the background; it is not interfering with the scheduler nor with low power mode. If that wasn't the case, then we wouldn't have seen any idle time at all because our tool issues an interrupt every 1.9ms. Another observation is that our performance analyzer was also able to detect itself. This means that once the performance analyzer issues an interrupt, the event handler does not block this interrupt and runs independently of following interrupts.

We can see that wireless communication requires a small percentage of time. This is for two reasons, first, the profiling data was being sent through the USART serial interface (or serially through USB) and not wirelessly. The second reason is that the application transmits/receives a packet at a slow rate. An important thing to observe is that the large percentage of idle resulted in a large number of wakeup calls which are known to be energy inefficient. Our main goal was to compare different operating systems, to understand how the difference in architecture and design will also lead to a difference in performance. Not only did we observe that, but we also found out that it also has an impact on building performance measurement tools as well. Depending on the software architecture or design philosophy used by the operating system, performance engineers have different capabilities in the embedded realm. For the comparison, we implemented the tool on Mantis and Contiki. Using the same hardware, we implemented the same timer wrapper discussed before. It is considered to be a port rather than an implementation. Porting the timer wrapper took several ours only, where as implementing it on TinyOS took several weeks. Figure 5 below show the results from Mantis and Contiki and their comparison with TinyOS' results, all three running our performance analyzer. The application running on the operating systems is the one described earlier. In Mantis the application is made up of two threads only, Rx and tx, the receiving thread and the sending thread. Collecting data from the sensors is implemented inside the sending thread. On Contiki, the application has two threads as well, one for sending, receiving and sensing and the other for topology management.

Before comparing the performance of these OSs let us focus on the results from Mantis and Contiki. As shown Figure 5, the results are less detailed as compared to the results from TinyOS. Each section in the pie chart resembles a thread in Mantis. Although a thread can call several functions; but it does not spawn threads. Functions called from within threads could be located using our tool but could not be accurately plotted. Let us explain more. In TinyOS, the snapshot taken by our tool is nothing but the current task's id. After collecting all the ids, we go to offline mode, where we map each id to its name using the app.c file generated at compile time. In TinyOS we were able to map 100% of the ids to their respective names. In Mantis however, the current running task's id is useless. This is because in offline mode, we don't have an app.c file generated and hence no means to map a task id with its name. We had to look for another approach. The best approach we found was able to map the name of the parent thread only, and not its inner function calls. The method we used on Mantis is as follows. The performance analyzer collects the address of the current running task instead of its id. The address will be sent to the base station and the offline analysis starts. To map an address to its name, we used the tool "msp430-objdump". This tool provided by the msp430 tool-chain, disassembles the compiled code (OS + application). Giving the tool the arguments as follows "map430-objdump --disassemble --source compiledapp.elf", generates a file containing memory addresses and machine language code intermixed with some high level source code. A snippet is illustrated below.

```
00004040 <tx_thread>:
{
    4040:  3f 40 2a 17      mov   #5930,     r15   ;#0x172a
    4044:  b0 12 cc 40      call  #16588           ;#0x40cc
    4048:  c2 4f 6e 17      mov.b r15,  &0x176e    ;

          404c: 3f 40 2a 17      mov   #5930,     r15   ;#0x172a
    4050:  b0 12 f4 52      call  #21236           ;#0x52f4

      //printf("[TX] sent %d bytes\n", send_buf.size);
          4054:  4f 43            clr.b r15        ;
    4056:  b0 12 5e 44      call  #17502           ;#0x445e

          405a: 3e 40 3f 00      mov   #63,  r14   ;#0x003f
    405e:  0f 43            clr   r15        ;
    4060:  b0 12 96 68      call  #26774           ;#0x6896
    4064:  ed 3f            jmp   $-36       ;abs 0x4040
  }
}
    4066:  30 41            ret
```

Although our tool was able to dissect Mantis much like TinyOS, the drawback came from insufficient information in offline mode for the offline analysis. For this reason, addresses such as 4056 had to be mapped to tx_thread even though the address might involve sensing and not transmitting. Porting the tool to Mantis took a few hours, but the offline analysis needs more time than TinyOS to be complete. This is a tradeoff in building the performance measurement tool for itself. If we go back to Figure 5, we see that our claim still stands. We are able to deduce that when running the same application, Mantis spent less time in idle mode. We are also able to deduce that this time was spent in the dispatcher (and in the application but mostly in the dispatcher). The dispatcher is the thread that has the sole purpose of removing a thread and placing another one on the stack. In other words, it does the context switch. This means that the reason why Mantis spends less idle time is that it is busy performing context switches. This can be directly inferred from our results and not by prediction as done by previous research. Moreover, the application also contributed into decreasing the idle time since it spent 36% of the time running as opposed to TinyOS' 23%

(everything other than idle and our tool). To identify exactly where that is happening, we need to further dissect the application.

Contiki is an event driven WSN OS. However, the goal behind Contiki was to overcome the problematic programming model associated with event driven OSs. So Contiki provides a thread driven programming model on top of an event driven kernel. The result is easy programming (unlike TinyOS); however in terms of performance analysis, the results are thread based. That is only threads could be monitored and the results are not expressive as TinyOS, instead they are more general like Mantis. This reflected by the results illustrated in Figure 5. Two questions now arise. First, can the tool measure network related metrics and other metrics as well? Second, what is the overhead of the performance analyzer?

To answer the first question, we further implemented another set of experiments, the first on TinyOS and the other on Mantis. We start with the first experiment. This experiment is intended to measure packet throughput, specifically receiving packet throughput. For that experiment we designed an application called "base station". This base station receives packets on the wireless interface and forwards them to the serial interface on a PC. We added a task to this application that is called every 250ms. The sending application on another node (not monitored) sends packets at the same rate. However we modified the task to be posted every 250ms to increase in size with time. We show how using our tool as is, without modifying it, we can measure packet throughput for this common application. As mentioned before, our tool generated task IDs. What we did is counted the number of ID's that are associated with receiving a packet. This is done offline to derive the number of received packets. From the interval of the experiment, we determined how many packets were received per second. As we have mentioned, the task length increases, each time by 10ms.

We plot that against the packets received per second, and we find out that as the task grows, packet receiving throughput decreases significantly. This is illustrated in Figure 6 below.

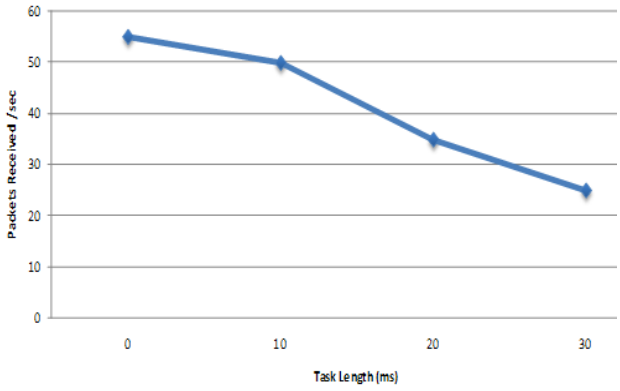


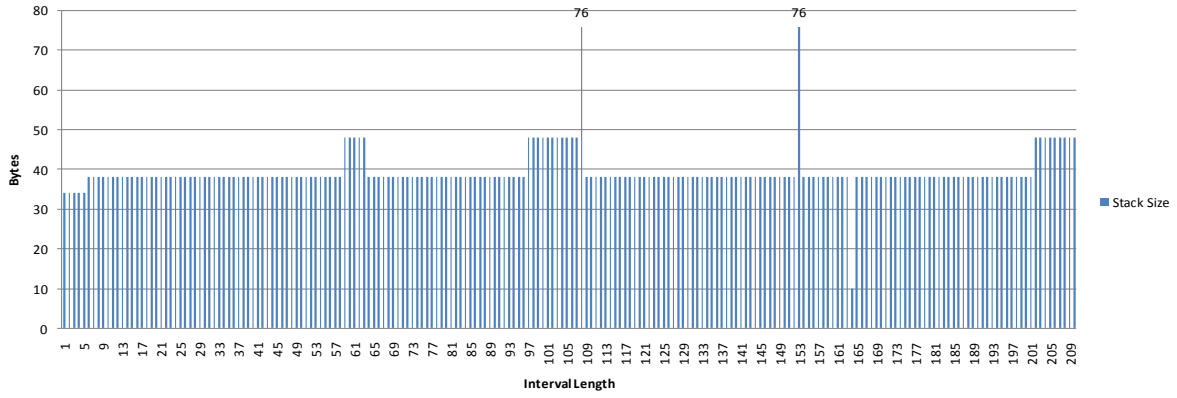
Figure 6. Packets processed decreases as task sizes increase

The explanation for this is as follows. Tasks in TinyOS are posted in a queue and execute din FIFO manner. For a task to affect receiving a packet, then the method receive must have a task posted on the same queue which is waiting for other tasks to finish. To find out whether this is the case, we tracked down the implementation of the event "Receive.receiveDone" and found out the whenever the low layer radio module decodes a packet, it posts a task that in turn signals the receive done event. Since this task is posted on the same queue with other tasks, the FIFO ordering will cause packet throughput to decline. We also observed that with tasks longer than 30ms, receiving packets halts completely (not shown in the diagram).

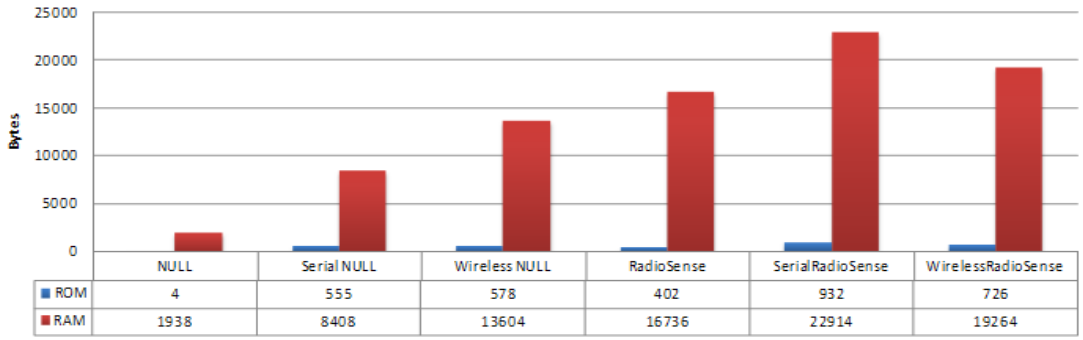
Mantis is a thread driven OS that requires a stack for each thread. When creating a thread in Mantis, the developer has to allocate the bytes for the stack himself. To aid the developer with stack usage, we upgraded our tool on mantis to collect not only the process address, but also how much stack has this process consumed. This is done by deducting the current value of the thread pointer from the total stack size. Both variables already provided by Mantis' scheduler. Figure 7 below illustrates the results highlighting the maximum amount of stack used by the application, which is lower than the maximum available (128). This way instead of allocating the maximum amount, 128 for every thread, some optimization can be made knowing that the application uses 76 bytes as a maximum. To answer the second question, we measure the overhead of our performance analyzer both on Mantis and TinyOS. In terms of Memory, the tool bsl (boot sector loader) used to load the compiled images onto the motes was used. Bsl tells us the Ram and Rom sizes of our image. Hence we loaded several images to identify the memory footprint of our analyzer. The first image called NULL was loaded to the mote. This image consists of nothing but TinyOS with no application. We deleted the image and loaded NULL with our tool added to it. Our tool uses 6466 extra bytes of RAM on top of NULL. This difference is decreased when adding our tool to TinyOS with an application. Figure 8 shows the same results for Mantis.

VI. Conclusions and Future Work

An important observation we have to mention is the huge involvement in machine code or low level assembly. The Timer wrapper for instance had to be implemented by manipulating registers on the microcontroller itself [30]. The event handler also involved translating machine code into Operating system specific logic in order to capture the interrupt generated from the hardware timer. Even in offline analysis when profiling Mantis, there is a great involvement in machine language as well. As a first step we have removed this burden from the shoulders of performance engineers and our tool provides a centralized location for measuring metrics in an energy efficient manner. We also need to classify our tool. Our tool is not an application, nor an operating system component, nor is it a piece of middle ware. Our tool lies under the class of daemons. A Daemon is a background process that is always running. Whether it reacts to events or does some processing itself depends on its purpose. This class of tools in wireless sensor networks is completely novel.



(a)



(b)

Figure 7. (a) Application Stack analysis using our performance analyzer on Mantis. (b) Memory footprint of our tool using TinyOS

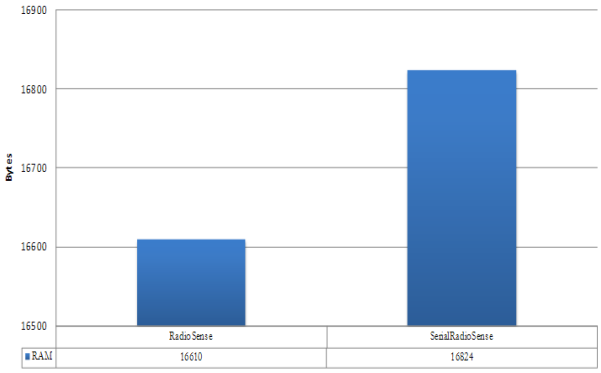


Figure 8. Memory footprint of our tool using Mantis

During our course of research, we have not yet seen any tool that can be classified as a wireless sensor node daemon process. This may be due to the fact that no tool has yet utilized the extra watchdog timer available as an interrupt generator. Hence we are introducing to the field a set of first timers that we believe will spawn a new spectrum of possible applications. Our aim is to make this tool a standard for WSN OS performance analysis. As a first step, we have already joined the TinyOS contribution society in order to allow the tool become available within future distributions of TinyOS. Once the tool is stable, the TinyOS community will assess it and might promote it to be included in all TinyOS distributions. This was the case with several tools in the field such as TinyDB, Surge and Deputy. Our TinyOS version of the tool is called TinyTune. Performance measurement is a very powerful technique when building new technologies. This paper fills the void of flexible, scalable, accurate and realistic performance measurement in the field of WSNs. We have built a tool for wireless sensor nodes that can be used to locate bottlenecks in the system. The tool can be extended to measure specific metrics such as context switches for example and other metrics as well. Our WSN Analyzer is built as a Daemon and can be distributed with new versions of the OS. We have also explained the phases required to build such a tool. Having this tool, we have set a new horizon for research in WSNs. Wireless Sensor Network OSs are known for their distinctive software architecture. Building such a tool for WSN EOSs introduced different challenges and a set of tradeoffs. Our aim is to build the same tool for other WSN EOSs such as SOS. We also intend to complement our statistical profiling tool with the call path profiling approach. Call path profiling has its own challenges and advantages as well. Moreover, our dissemination technique is trivial. Messages are broadcasted to reach the base station. A more efficient Dissemination technique could be implemented as a next step. Several dissemination techniques could be analyzed using our tool to decide upon. Our WSN Analyzer collects data online and measures them offline. However we can make use of the online information to make the network react on the spot. TinyOS for example has a set of programming guidelines or rules (written by Philip Levis) that when followed, produce a more efficient and reliable system. Such rules cannot be enforced by the nesC compiler for flexibility; however, our tool could be extended to notify the programmer, at runtime, if any of these rules have been violated. This way our tool could act as an add-on to the nesC compiler. As for offline

measurement, we intend to develop a powerful GUI that can plot the results automatically. For example, a user can click on a section in the pie chart to further dissect it or view its source code.

ACKNOWLEDGMENT

The authors wish to thank the National Research foundation for their support. This work was partially supported by the National Research Foundation Grant No. 0536/2012.

REFERENCES

[1] Hill, J., R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. "System Architecture Directions for Networked Sensors." *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems. New York, USA: ACM Press,* November 2000, 93-104.

[2] Karl, H. and A. Willig. *Protocols and Architectures for Wireless Sensor Networks. New York: John Wiley & Sons,* April 2005, 45-50.

[3] Han, C., R. Kamur, R. Shea, E. Kohler, and M. Srivastava. "A dynamic operating system for sensor nodes." *Proceedings of the third international conference on Mobile systems, applications, and services. New York, USA: ACM Press,* June 2005, 163-176.

[4] Bhatti, S. J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth. B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. "MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms." *ACMKluwer Mobile Networks and Applications Journal, Special Issue on Wireless Sensor Networks. Hingham, USA: Kluer Academic Publishers,* August 2005, 563-579.

[5] Duffy, C., U. Roedig, G. Herbert, and C. Sreenan. "An Experimental Comparison of Event Driven and Multi-Threaded Sensor Node Operator systems." *Proceedings of the fifth Annual IEEE International Conference on Pervasive Computing and Communications Workshops. White Plains, New York, USA: IEEE computer society,* March 2007, 267-271.

[6] Kim, H. and H. Cha. "Multithreading optimization techniques for sensor network operating systems." *Wireless Sensor Networks. Heidelberg, Berlin: Springer,* April 2007, 293-308.

[7] Hujumg, C., C. Sukwon, J. Inuk, K. Hyoseung, S. Hyojeong, Y. Jaehyun, Y. Chanmin. "RETOS: Resilient, Expandable, and Threaded Operating System for Wireless Sensor Networks." *Proceedings of the sixth international conference on Information processing in sensor networks. Massachusetts, USA,* April 2007, 148-157.

[8] Dunkels, A., B. Gronvall, T. Voigt, "Contiki – A Lightweight and Flexible Operating System for Tiny Networked Sensors." *Proceedings of the twenty ninth Annual IEEE International Conference on Local Computer Networks,* November 2004, 455-462.

[9] Duffy, C., U. Roedig, G. Herbert, and C. Sreenan. "A performance analysis of mantis and tinyos." *University College Cork, Ireland, Technical Report CS-2006-27-11,* November 2006.

[10] Gay, D., P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. "The nesC Language: A Holistic Approach to Networked Embedded Systems." *Proceedings of Programming Language Design and Implementation.* California, USA, June 2003, 1-11.

[11] Multimodal Networks of In-situ Sensors; available from http://mantis.cs. colorado.edu; Internet; accessed 20 March 2008.

[12] Welsh, M., and R. Newton. "Region Streams: Functional Macroprogramming for Sensor Networks." *Proceedings of the first workshop on data management for sensor networks. Toronto, Canada,* August 2004.

[13] Behren, R., J. Condit, and E. Brewer. "Why Events Are A Bad Idea (for high-concurrency servers)." *Proceedings of HotOS IX: The ninth Workshop on Hot Topics in Operating Systems. Hawaii, USA: USENIX Association,* May 2003, 19-24.

[14] Lauer, H. and R. Needham. "On the Duality of Operating System Structures." *Proceedings of the second international Symposium on Operating Systems. Rocquencourt, France: IR1A, October 1978; reprinted in Operating Systems Review* (April 1979): 3-19.

[15] Gustafsson. "Threads Without the Pain." *New York, USA: Queue, ACM Press,* November 2005, 34-41.

[16] Yi, S., H. Min, J. Heo, B. Boand, and E.F. Roberts. "Performance analysis of task schedulers in operating systems for wireless sensor networks." *Computational Science and its Applications.* Heidelberg, Berlin:Springer, May 2006, 499-508.

[17] TinyOS; available from http://www.tinyos.net; Internet; accessed 20 March 2008.

[18] Rhee, A. Warrier, M. Aia, and J. Min. "Z-MAC: a hybrid MAC for wireless sensor networks." *Proceedings of the third international conference on embedded networked sensor systems. New York, USA: ACM Press,* November 2005, 90-101.

[19] Pompili, D. and T. Melodia. "Three-dimensional routing in underwater acoustic sensor networks." *Proceedings of the second ACM international workshop on Performance evaluation of wireless ad hoc, sensor, and ubiquitous networks. New York, USA: ACM Press,* October 2005, 214-221.

[20] Watfa, M. "Wireless Sensor Networks, Making a Difference Tomorrow." *Proceedings of the fourth International IEEE Conference on Ubiquitous Intelligence and Computing. Lecture Notes in Computer Science, Springer, Verlag, Hong Kong,* July 2007.

[21] Lee, Y., F. Cheng, and Y. Leung. "Exploring the impact of RFID on supply chain dynamics." *Proceedings of the thirty sixth conference Winter simulation. Winter Simulation Conference, Washington, USA, December 2004,* 1145-1152.

[22] Lindsley, R. "Kernel Korner: What's new in the 2.6 Scheduler." *Linux Journal Specialized Systems Consultants, Inc, Seattle, USA,* March 2004, 13.

[23] Bartal, Y., S. Leonardi, G. Shallom, and R. Sitters. "On the value of preemption in scheduling." *Approximation Randomization, and combinational Optimization. Algorithms and Techniques.* Springer, Heidelberg, Berlin, August 2006, 39-48.

[24] Tanenbaum, A. and A. Woodhull. Operating systems design and implementation. *New York: Prentice Hall,* 2006, 33.

[25] Ousterhout, J. "Why threads are a bad idea (for most purposes)." *Presented at the 1996 Usenix Annual Technical Conference, San Diego, USA,* January 1996.

[26] Duffy, C., U. Roedig, G. Herbert, and C. Sreenan. "Improving the Energy Efficiency of the MANTIS Kernel." *Proceedings of the fourth IEEE European Workshop on Wireless Sensor Networks. Delft, Netherlands: IEEE Computer Society Press, January* 2007.

[27] Moubarak, M. and M. Watfa. "Optimizing the Value of Preemption in Embedded Wireless Sensor Nodes." *Proceedings of the International conference on Embedded Systems and Applications, ESA 08'*, Las Vegas, 2008.
[28] Duffy, C., U. Roedig, G. Herbert, and C. Sreenan. "Adding Preemption to TinyOS." *Proceedings of the fourth workshop on embedded networked sensors, Cork, Ireland: ACM Press*, June 2007.
[29] Singhania, A., C. Han, and B. Srivastava. "Knots: An Efficient Single Stack Preemption Mechanism for Resource Constrained Devices." *University of California, Los Angeles, Technical Report UCLA-NESL-200710-02*, October 2007.
[30] MSP430 Manual at Texas Instruments site; available from http://www.ti.com/; Internet; accessed 20 March 2008.
[31] John, L. and L. Eckhout. *Performance Evaluation and Benchmarking. Florida, Taylor and Francis group*, 2006, 1-282.
[32] EDN Embedded Microprocessor Benchmarking Consortium; available from http://www.eembc.org/Internet; accessed 20 March 2008.
[33] Guthaus, M., J. Ringenberg, D. Ernst, T. Austin, T. Mudge and R. Brown. "MiBench: A free, commercially representative embedded benchmark suite." *IEEE International Workshop on Workload Characterization, Michigan, USA, December* 2001, 3-14.
[34] Brown, A. "A Decompositional Approach to Performance Evaluation." *Technical Report TR-03-97, Center for Research in Computing Technology, Harvard University*, 1997.
[35] Park, S., J. Kim, K. Shin and D. Kim. "A Nano Operating System for Wireless Sensor Networks." *8th International Conference on Advanced Communication Technology*, February 2006, 345-348, 51.
[36] Moubarak, M. and M. Watfa. "Embedded Operating Systems in Wireless Sensor Networks." To appear in *Handbook of Wireless Ad-hoc and Sensor Networks*, London, Springer, 2008.
[37] Koopman, P., J. Sung, C. Dingman, D. Siewiorik and T. Marz. "Comparing Operating Systems Using Robustness Benchmarks." *In Proceedings of the sixteenth Symposium on Reliable Distributed Systems*, October 1997, 72-79.
[38] Lee, C., M. Potkonjak and H. Mangione-Smith. "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems." *Micro-30, November* 1997, 330-335.
[39] Wright, C., N. Joukov, D. Kulkarni, Y. Miretsky and E. Zadok. "Auto-pilot: A Platform for System Software Benchmarking." *In Proceedings of the Annual USENIX Technical Conference*, CA, April 2005, 175–187.
[40] Li, T., L. Kurian John, A. Sivasubramaniam , N. Vijaykrishnan and J. Rubio. "Understanding and improving operating system effects in control flow prediction." *Proceedings of the tenth international conference on Architectural support for programming languages and operating systems*, California, October 2002, 60-68.
[41] Kucuk, G. and C. Basaran. "Reducing Energy Consumption of Wireless Sensor Networks through Processor Optimizations." *Journal of Computers*, July 2007, 67-74.
[42] Law, Y., J. Doumen and P. Hartel. "Survey and benchmark of block ciphers for wireless sensor networks." *ACM Transactions on Sensor Networks*, February 2006, 65-93.
[43] John, L. "More on finding a single number to indicate overall performance of a benchmark suite." *ACM SIGARCH Computer Architecture News*, March 2004, 3-8.
[44] Nguyet, T., M. Nguyen and Mary Lou Soffa. "Program Representations for Testing Wireless Sensor Network Applications." *DoSTA, Croatia*, 2007.
[45] Hempstead, Mark, Matt Welsh, David Brooks. "TinyBench: The Case for A Standardized Benchmark Suite for TinyOS Based Wireless Sensor Network Devices." *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN'04),* October 2004, Switzerland.
[46] Nazhandali, Leyla, Michael Minuth, Todd Austin. "SenseBench: Toward an Accurate Evaluation of Sensor Network Processors." *Proceedings of the IEEE International Workload Characterization Symposium, October 2005, Austin, Texas, USA.*
[47] Shnayder, Victor, Mark Hempstead, Borrong Chen, Geoff Werner Allen, and Matt Welsh. "Simulating the Power Consumption of LargeScale Sensor Network Applications." *SenSys'04, November 2004, Baltimore, Maryland, USA.*

Mohamed K. Watfa is currently an Associate Professor in the Faculty of Computer Science and Engineering. Prior to this, he was an Assistant Professor at the American University of Beirut (AUB). At 24, Dr Watfa was one of the youngest PhD holders to graduate from his university. Dr Watfa's research interests include wireless and computer networks, wireless sensor networks, Vehicular Adhoc Networks (VANETs), intelligent and ubiquitous systems, network security, and energy efficient protocols. He has more than 50 journal and conference publications ranked among the top in his research domain. Dr Watfa received the competitive UOWD Research Excellence Award which was a direct result of his ambitious research track record in 2009. He also won the same excellence award again in 2011 along with the Teaching Excellence Award. He was also shortlisted in the prestigious Rolex Award for Enterprise in 2011. Aside from this, he has also been granted a number of national research grants including an Emirates Foundation Research Award and the National Research Foundation Grant in 2012. Dr Watfa is a professional member of the ACM and IEEE. For more information on Dr Watfa, please visit his personal website at: http://mohamedwatfa.synthasite.com

Mohamed Moubarak finished his Masters in Computer Science from the American University of Beirut (AUB) after working on a number of research projects under the supervision of Dr. Watfa. His research interests include sensor networks and tiny operating systems benchmarking methodologies. He is a professional member of the ACM and IEEE and currently invovled in number of projects in the RFIDS applied sector.