1-1-2008

Optimizing the value of preemption in embedded sensor nodes

A Mohamed Moubarak
American University of Beirut

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

Optimizing the Value of Preemption in Embedded Sensor Nodes

A. Mohamed Moubarak [1], **and B. Mohamed Watfa** [1]
[1] Department of Computer Science, American University of Beirut, Beirut, Lebanon

Abstract - *The emergence of the technology of Wireless Sensor Networks has lead to many changes in current and traditional computational techniques. Traditional operating systems do not take into consideration the limitations in space and energy of wireless sensor networks. New system architectures have emerged to overcome these limitations. Each follows one of two design concepts, event-driven or thread-driven. This paper studies the differences between the aforementioned system designs, pointing out the causes of the tradeoff. The paper then introduces a thread-driven scheduling algorithm focusing on the value of preemption to overcome the energy tradeoff brought by event-driven systems. Our proposed algorithm reduces the average amount of energy spent under high system load which was a significant scenario where event-driven systems showed better energy savings.*

Keywords: Sensor Networks, Operating Systems, Multi-threading, Energy Efficiency.

1 Introduction

New revolutionary protocols and algorithms in the field of networking have appeared due to the limitations of wireless sensor networks. An interesting field that was also affected by the advancement of wireless sensor networks is that of operating systems. Wireless sensor networks are expected to run a variety of sensing applications, reading in all types of data from acoustic to temperature values. These sensors will need to do some pattern recognition, after which the sensors will diffuse data on to a non manageable network. This requires the running of applications ranging from location aware algorithms to energy efficient routing. This is just a glimpse of what these sensors should handle. To make the transition from what these sensors should handle to what they actually could handle, a new type of operating system is needed to manage all the resources and applications. This operating system has to do so taking into consideration security, energy efficiency, high concurrency and extremely low memory. This sounds like a blend of three types of systems that exist today, personal computers, distributed systems, and real time systems. The required operating system is also required to run on an 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]. This kind of architecture introduces an entire field of challenges. The desired operating system will directly affect the performance of individual applications and even the entire network of wireless sensors.

Existing operating systems do not meet these requirements and hence the work on applicable operating systems has begun. The de facto operating system for wireless sensors is TinyOS [1]. Another embedded operating system designed for wireless sensors is MOS [3]. Unlike TinyOS, MOS is thread-driven. That is, tasks are preempted by the scheduler for other (higher priority) tasks to run. This provides the aspect of virtualization desired in operating systems.

Although other operating systems also exist in the field such as SOS [4], all operating systems conform to one of two design philosophies, event-driven and thread-driven. The choice of which design to adopt is not made abruptly, instead, it is thoroughly investigated since it has a significant impact on the performance of the remaining system in its remaining life time. The importance of choosing among an event-driven system and a thread-driven one has motivated us to contribute to the field. Any application, algorithm or protocol will have to conform to the chosen design, hence carrying with it the design's advantages and disadvantages. Making the choice at an early stage obliges the designer to go back to existing results of prior experiences and theoretical analysis. Event-driven systems are assumed to perform better under constrained environments. Yet they lack some system functionality and impose their own difficulties. However, thread-driven systems provide high concurrency with preemption, allowing the use of real-time applications.

This paper presents a thread-driven optimization technique focusing on the value of preemption. The technique aims at conserving more energy, thus overcoming the tradeoff that was pointed out by previous research. Our algorithm is implemented and its performance evaluated. Our results show a decrease in the number of context switches along with an increase in idle time, and thus improving the energy efficiency of thread-driven systems. The rest of this paper is organized as follows. Section 2 presents related work. Section 3 describes the notions of event-driven and thread-driven systems. Section 4 investigates the differences between each model. Section 5 describes the proposed thread-driven optimization algorithm, evaluation environment and implementation specifics. Section 7 presents the results obtained from experimentation. Section 8 concludes the paper and discusses some future work.

2 Related Work

In [9], the authors make a first attempt at optimizing the low level implementation of thread-driven operating systems, in order to achieve event-driven performance. First, the authors perform stack analysis and used control flow information created at compile time to predict the size of the stack. Then, they provided a single stack implementation for all running threads, as opposed to the traditional technique of creating a stack for each thread, thus cutting down on space. The authors also tackle energy consumption by coming up with a new scheduling technique that depends on a variable timer, as opposed to the traditional fixed quantum, thus saving on computation latency. However, they did not take into account the large overhead produced by context switches. Their results still perform worse than event-driven systems, but with a great improvement compared to other thread-driven systems.

Our work is greatly motivated and influenced by the works of [10] and [2]. In [10], the authors make a first step in studying the cost of preemption. The authors present a theoretical scheduling model which incorporates the cost of preemption. They show that preemptive algorithms, such as shortest remaining processing time, are theoretically optimal but are impractical because they do not take into consideration the cost of context switches. Moreover, the authors provide an algorithm, "wait to preempt", which aggregates arriving processes and then runs them after a certain amount of work is done, which depends on the cost of preemption. However the authors aim at minimizing total flow time, which is the total time that the jobs spend in the system since arrival until they are run to completion. The cost of preemption introduced does not depend on energy consumed or on the CPU cycles. The algorithm is strictly based on the size of processes and also assumes the knowledge of the size of the smallest process.

The authors in [2] comparatively evaluate the performance of MOS and TinyOS. Their work measures the memory foot-print, event processing and energy efficiency of the two operating systems. The experiments aimed at comparing the performance of event-driven systems against thread-driven ones. The results show that the event-driven system, specifically TinyOS, has smaller memory foot-print and better energy consumption at high system loads. Whereas the thread-driven MOS has better real time performance and predictability with similar energy consumption at low system loads. According to these results, a tradeoff exists when choosing among those systems.

The same authors in [2] attempted to overcome this tradeoff later on in [13] and [14]. In [13], the authors focus on improving energy efficiency in MOS by tuning its preemptive scheduler. Their modifications included removing the idle thread, which ran whenever no tasks are runnable. Also, time slicing between equally prioritized threads was removed. If needed, the user should explicitly include it. Finally the linked list queues were replaced by a single array, which makes addition and deletion costly. This tuning technique is specific to MOS and not to thread-driven systems like ours; however it improves the energy efficiency of MOS.

3 Events and Threads

Before investigating the difference between the event-driven design and the thread-driven one, we will describe the two designs. Event-driven models consist of event handlers that continuously wait for events to issue tasks such as packet arrivals to be processed. Since tasks may arrive at a pace faster than that of the processor, tasks are queued. The scheduler of the event-driven model selects the tasks from the queue to be processed in a FIFO fashion. The selected task is then put on the processor and processed to completion, uninterrupted by other tasks. After the completion of the entire task, the scheduler can select the next task to process and so on.

Thread-driven systems on the other hand deal with tasks in a different way. When a task is created, it is queued. The scheduler selects a thread from the queue in any fashion; let us assume a round robin scheduler, like the one in MOS. The thread is put on the processor for a certain time slot after which the thread is preempted (interrupted) and another thread is put on the processor. By allowing multiple threads to execute preemptively, the system acts as if there are multiple processors, one for each thread. The next section elaborates more on the difference between the two design philosophies.

4 Events vs. Threads

Event-driven programming has been highly advertized in recent years as the best way to approach concurrent applications [5]. However, after more research has been done, it has been shown that the latter belief is not completely true. The arguments in favor of the event-driven model are that it uses an inexpensive (non-preemptive) scheduling technique, it requires no stack management and provides a safe control flow (no locks and semaphores) [5]. Moreover, event-driven systems are highly portable since they do not require the extra stack support for multi-threading. They also have a smaller memory stamp [2] [6]. However, in [7], the authors have shown that event-driven systems could still have the same performance of thread-driven systems.

4.1 Programmer Experience

According to [8], event programming is tedious, unstructured, and repetitive. In the event-driven design, the event loop is in control and not the programmer. So, the programmer will have to chop a program into a series of short programs. This is also required in order not to allow a long running task to monopolize the entire system. However, in a thread-driven implementation, the programmer is not concerned whether his program monopolizes the system or not, since the system itself will take care of that through its preemptive nature.

4.2 Bounded Buffer Producer-Consumer Problem

Due to the RAM limitations in embedded wireless sensors, the buffers are sufficiently small for the bounded buffer producer-consumer problem to occur in an event-driven system. When an event is filling up a buffer in an event-driven system, the buffer will not be emptied by a consumer unless the current event or the producer is done putting all the data it got on to the buffer. The buffer may be full for a time long enough to lose data such as packets that could not find space in the buffer. However, in a preemptive or thread-driven system, the buffer will be occasionally emptied by other events running virtually in parallel, avoiding the problem of producer-consumer bounded buffer. In event-driven systems, long lived tasks may exist under high system load due to the complexity of applications running [3].

4.3 Disadvantages of Preemption

Preemption has played an important role in drawing the line between event-driven systems and thread-driven ones. Several research papers show that all the fears of multi-threading comes from preemption [2] [8]. To elaborate, let us look at the disadvantages of the thread-driven approach.

One argument against the thread-driven approach is the difficulty in writing code that handles synchronization through semaphores or monitors [8] [12]. The reason why locks are needed as a form of synchronization is because threads may be using shared variables while they run preemptively. Thus, as in [8], the question whether the control flow is event-driven or thread-driven is orthogonal to the question of whether those threads and events were preemptively scheduled.

To illustrate the motivation behind our work, we present some experiments that compare the performance of TinyOS and MOS under high system load. Experiments comparing TinyOS and MOS have shown that under high system load, MOS consumes more energy. In these experiments, a tree binary topology is assumed. Depending on the tree position *n* in the tree, a sensor node might process varying amounts of packets. The behavior of a single node is emulated by applying a certain traffic pattern. The node under test was given varying sensing task lengths and a set of forwarding tasks to emulate each tree position *n*, hence each node was stressed depending on whether it is a leaf node or a forwarding node. The idle time was measured at every position *n* in the tree as an indication of the amount of energy conserved, as illustrated in Fig. 1. The difference in idle time is directly related to context switches or preemption, since under high system load, the number of incoming packets increases the number of interrupts. Under low system load, MOS offers better concurrency, prediction, and equal energy consumption as the event-driven TinyOS.

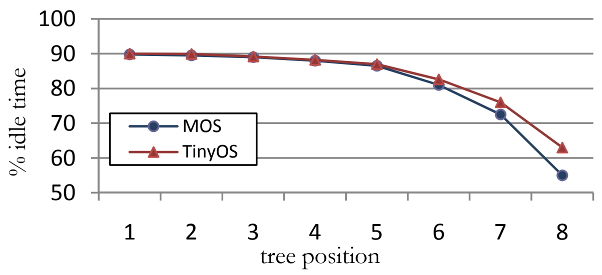
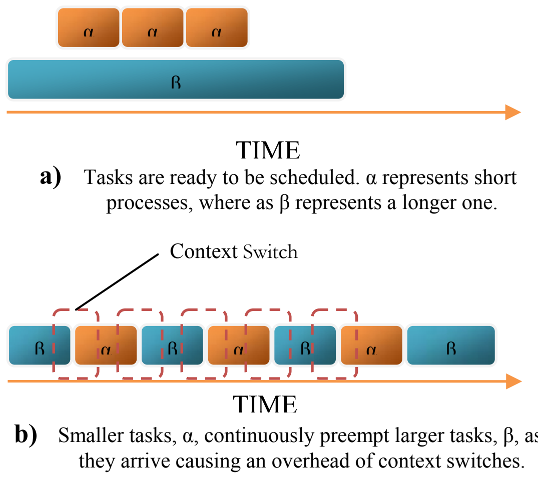


Fig. 1. Plots of idle time with increasing sensing length showing that TinyOS is more energy efficient than MOS under high loads

5 Optimized Preemption

Now that we have seen that the main fears of multi-threading come from the value of preemption, we tackle this problem by introducing an energy efficient preemption optimization. Our algorithm aims at optimizing the number of context switches in thread-driven systems, under high system loads. This is done by directly optimizing the number of preemptions. There are two scenarios that need to be taken into consideration under high system load. First, when sensing tasks are timely. When smaller tasks arrive, the longer sensing task will be continuously preempted (Fig. 2). This causes preemption overhead, and is worse when tasks are longer. The second scenario does not involve the size of incoming tasks; instead it involves the frequency at which they arrive. At high frequencies, processes tend to preempt each other irrelative of their sizes.



a) Tasks are ready to be scheduled. α represents short processes, where as β represents a longer one.

b) Smaller tasks, α, continuously preempt larger tasks, β, as they arrive causing an overhead of context switches.

Fig. 2. Without taking into consideration the size of the process, scheduling may cause context switch overhead

Taking these scenarios into consideration, our algorithm works as follows. First, run processes preemptively in a round robin fashion. After some work δ has been done, preempt the currently running process if it is long, and run small processes to completion without preemption. Again after some work has been done, go back to step one of the algorithm and repeat. The algorithm presented depends on three values, α, β and δ. α represents the size of a small process, β the size of a long process and δ denotes a certain amount of work done. The idea (illustrated in Fig. 3) is to create preemption free periods without affecting concurrency by differing small processes and running them to completion. The following sections elaborate on the choice of α, β and δ.

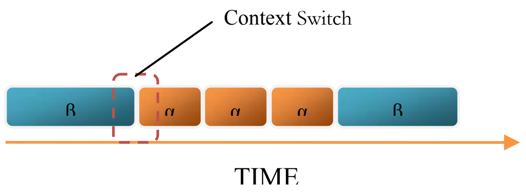


Fig. 3. Using our algorithm, only one context switch is needed in the same scenario of Fig. 2-b

5.1 Process Sizes α and β

Accurately determining the size of a process is almost impossible yet is a very crucial piece of information. Several scheduling algorithms used in the field depend on the size of a process. One approach to predict the size of the process is called aging [11]. The size of a process depends on the amount of time it has spent on the CPU during previous runs. Hence the update is continuously updated. Formally, assume a process spent time T_0 on the first run and T_1 on the second run. The new estimate is the weighted sum of these two runs, that is $aT_0 + (1 - a)T_1$, where a is the chosen weight. However our approach in determining the size of a process is simpler and is based on the quantum size. α and β are discussed in more detail in section 5.3.

5.2 Work Done δ

The proposed algorithm mainly depends on the value δ. The value δ denotes the time when the scheduling algorithm should adapt to optimize the number of context switches. This is done by the scheduler entering a preemption free period. In this period, small processes are run to completion with respect to each other. This is because small processes are handled quickly and easily. After another δ, the scheduler returns to its original state, allowing longer processes to run. The algorithm is illustrated in Fig. 4. The value δ could be tuned for better performance and could be determined based on experimentation. Our choice of δ is discussed in the following section.
Using this approach, we might incur some delay in terms of the amount of time processes wait to be scheduled. To optimize this latency, one method that can be used to increase latency is by enhancing the CPU utilization. Such approach is presented in [9] and works as follows. When the clock interrupt handler determines the end of a quantum a context switch occurs. However the clock will keep issuing interrupts at a certain rate. Since most of these interrupts are unhandled, a considerable amount of energy is wasted in triggering them. To overcome this problem, a variable timer was implemented such that the rate at which interrupts occur depends on an upcoming timeout request. The variable timer in [9] manages timeout requests from threads and sets the clock-tick rate as such. Variable timers are not feasible in conventional OSs where the number of threads is very large. However, in networked nodes, the number of threads is small enough to allow for a variable timer.

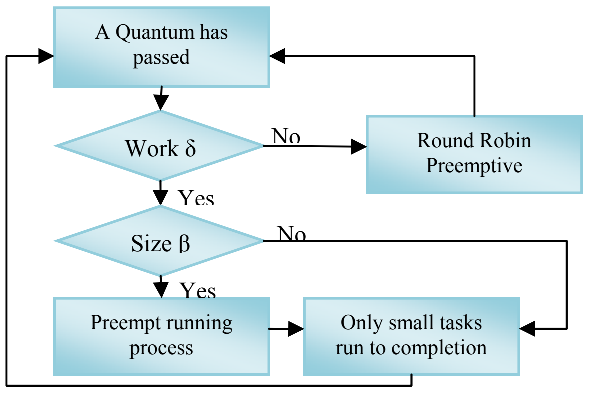


Fig. 4. After each quantum, check if a certain amount of work δ is done. If so, check if the running process is long (β). If so, preempt it and run only small processes (α) to completion without preemption

5.3 Implementation

In this section, we discuss implementation specifics, namely the choices of the values α, β and δ. Before doing so, we need to present the two different types of context switches, voluntary and involuntary. A voluntary context switch occurs when a job or process gives up its time quantum voluntarily due to an IO request for example. An involuntary context switch on the other hand is when a process uses up its quantum but still has work to do. In this case the kernel preempts the process to place another one. We are only interested in optimizing the value of involuntary context switches. We mentioned previously that we use the quantum to determine the size of a process.

This is done as follows. On each clock tick, the kernel checks if the current process has used up its quantum. Processes are given a fixed quantum and are not preempted before the quantum is done. A process may require more than one quantum to finish. So if the kernel determines the end of the current process' quantum, the kernel will preempt the process causing an involuntary context switch. The scheduler will place the preempted process in the appropriate place in the scheduling queue and pick another process to run. When a process is preempted for an IO request, the quantum that it used is recorded. So when the process gets its request and is put back on the CPU, it is not given a full quantum again. It is only given the remaining quantum it had left. However, if the process was preempted due to an involuntary context switch, it is given a full quantum again (illustrated in Fig. 5). Thus we have the notion of a small process and a large process depending on the remaining quantum size. More precisely, if a process has a full quantum, it's a long process β; otherwise it's a short process α.

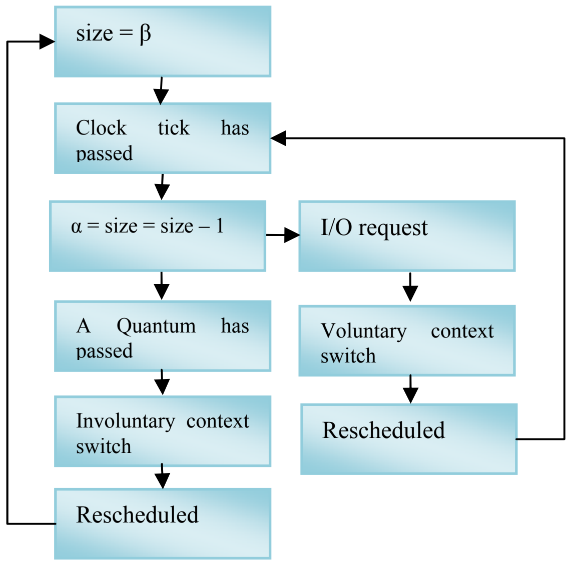


Fig. 5. Short and long processes α and β identified by quantum size

As for the value of δ, we represent the work done in terms of time spent. Another research effort represents the work done as a ratio of preemption cost and the size of the smallest task [10]. However, for the sake of simplicity, we use the value of δ to be 100 quanta. In other words, every 100 quanta, the scheduler readapts to optimize preemption.

5.4 Benchmark Suite

To evaluate the performance of our optimized scheduler, we have implemented a benchmark suite that simulates a system under high load. Our benchmark assumes a tree topology. Nodes with larger height *h*, have more work to process, while nodes with lower *h* are less loaded. To simulate the load relative to the position in the tree, the benchmark uses two variables, the frequency f_s at which packets arrive and the sensing duration l_s. By varying these values, the position h_i in the routing tree is simulated. In our simulation, we are only interested in nodes that experience high system loads, illustrated in Fig. 6. This is because the overhead of context switches only appear then. In our benchmark, high system load is represented by values of f_s and l_s being 300000 CPU cycles and 1000 *ms* respectively. Moreover, 4 copies of the benchmark were run at once, to simulate the existence of 4 neighboring nodes.

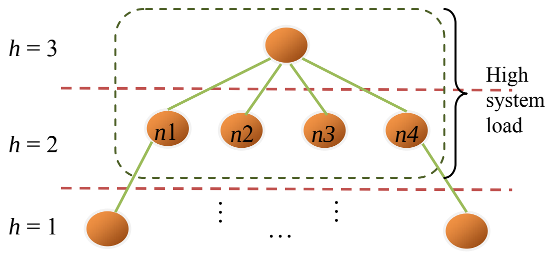


Fig. 6. Network routing topology forming a tree. The greater the height h, the closer the node is to the sink or the root. The high system load area is the area of interest.

6 Results

Our benchmarking suite was run for one minute before and after implementing our scheduling algorithm. The performance of the system was monitored and plotted to show the change in energy consumption and the affect on event processing.

6.1 Energy Consumption

We have shown in previous sections the effect of context switches on the energy efficiency of a system. The more the context switches, the more energy is consumed. We argue that if we decrease the number of context switches while still doing the same amount of work, we obtain better energy consumption. From the OS perspective, energy is not measured by the amount of current dissipated, instead it is measured by idle time. The energy efficiency of an OS is how much it can provide idle time for the CPU. By sparing the CPU some of its cycles, the result is better energy consumption. In the first experiment, the number of CPU cycles spent is plotted before and after our implementation.

The results illustrated in Fig. 7 are an indication of idle time. The amount of CPU cycles spent after our optimization is less than those spent without it. This is because we reduced the number of context switches and therefore reduced the total amount of processing the CPU has to perform. In the time frame of the experiment, the same amount of packets was delivered before and after, and the same length of sensing tasks as well. Yet, due to the reduction in the number of times the CPU has to switch between processes, the CPU does less work. This is a direct indication of both idle time and energy consumption, i.e. the less the cycles, the more the CPU is idle and the more energy is conserved.

In the second experiment, the total number of context switches is monitored. As mentioned earlier, we simulate packets coming from 4 different neighbors. The amount of processing done for each neighbor is monitored and the number of context switches is calculated as well. In Fig. 8, the number of context switches due to each neighbor is plotted before and after our optimization. A significant decrease in the number of context switches is shown due to our optimization. This is expected since our algorithm is able to reduce context switches by more than 70 percent. That is the total number of context switches due to processing packets coming from all neighbors.

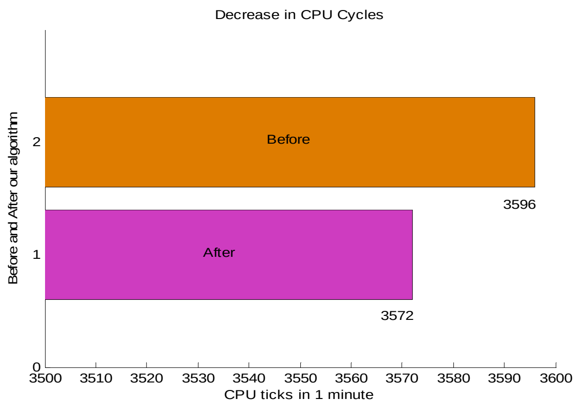


Fig. 7. Number of CPU ticks decreased using our algorithm since the number of context switches has been optimized. Using our algorithm the CPU has more idle time and hence consumes less energy

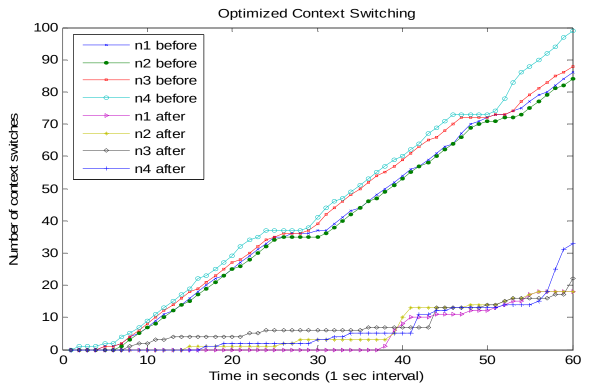


Fig. 8. The number of context switches is optimized using our scheduling algorithm

6.2 Event Processing

Although we have optimized preemption, this was expected to incur an overhead in terms of delay. Our next experiments investigate this delay and its effect on event processing. Fig. 9 presents the effect of our optimization on the predictability or real-time operation of the system. The average processing time is calculated and plotted before and after our optimization. The average is the total processing time spent for all neighbors divided by the number of neighbors. The delay incurred by our algorithm hence would be the difference between the average processing time before and after. As shown in the plot this difference is very small, hence event processing is slightly affected. This delay is affected by the choice of the parameter δ discussed in Section 5.

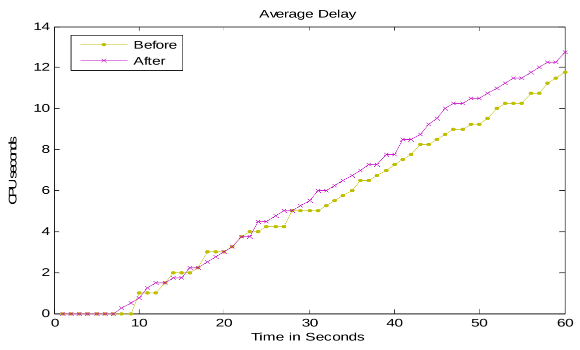


Fig. 9. Event processing is slightly affected by our optimization

We were also interested in investigating the relation between the size of processes and behavior of context switches. As the number of long processes increases, the number of context switches is expected to increase. Moreover, our algorithm has more potential for conserving energy when there are enough small processes to run without preempting longer tasks. For example, if the number of short processes is small, the scheduler will go back to its default (round-robin) state before the amount of work δ has been done. Otherwise the scheduler will cause a deadlock. If small processes cannot cover the period δ, the scheduler will be running long and short processes as if it is a round-robin scheduler since it will always go back to its default state. However, we know this is not often the case at high system load. This is illustrated in Fig. 10. The number of context switches increases steadily and at a low rate as small processes arrive. At time = 40 sec, a significant decrease in the number of short processes causes a rapid increase in the number of context switches. The plot also shows that the percentage of small processes is not very high. This means that the number of voluntary context switches is low, and the overhead is due to involuntary context switches. Since short processes have smaller quanta, processes that perform voluntary context switches are fewer. This is because a smaller quantum is a result of a voluntary context switch in the first place. Hence voluntary context switches do not dominate the overhead of preemption which justifies our focus on involuntary context switches.

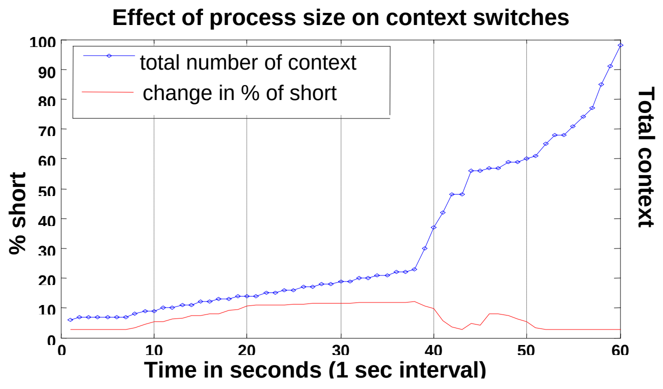


Fig. 10. The relation between percentage of short processes and context switch behavior

7 Conclusion and Future Work

In this paper, we show 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. As part of our future work, we are to provide a deeper investigation on the effect of our algorithm on processing latency. We also intend to investigate different values for δ and its effect on delay. A theoretical analysis of our algorithm would be provided in an extended version of this paper. An investigation involving more Wireless sensor OSs is required to determine other bottlenecks.

8 References

[1] J. Hill, 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* , ACM Press, New York, USA, November 2000, pp. 93-104.

[2] C. Duffy, 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*, IEEE computer society, White Plains, New York, USA, March 2007, pp. 267-271.

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

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

[5] R. Behren, 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* , USENIX Association, Hawaii, USA, May 2003, pp. 19-24.

[6] C. Duffy, 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.

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

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

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

[10] Y. Bartal, 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, pp. 39-48.

[11] A. Tanenbaum and A. Woodhull, *Operating systems design and implementation*, Prentice Hall, 2006.

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

[13] C. Duffy, 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*, IEEE Computer Society Press, Delft, Netherlands, January 2007.

[14] C. Duffy, U. Roedig, G. Herbert, and C. Sreenan, "Adding Preemption to TinyOS" *Proceedings of the fourth workshop on embedded networked sensors*, ACM Press, Cork, Ireland, June 2007.