DECOUPLING APPLICATION PARAMETERS IN WSN AND IMPLEMENTATION OF Y-THREADS IN LITEOS

Debojit Dhar, Maliha Sultana and Sriram Murali

Department of Electrical and Computer Engineering, University of British Columbia

ABSTRACT

This paper proposes methodologies for better utilization of sensor nodes for large scale applications that requires good time sensitivity, memory utilization and adaptability. In real-time applications such as motion sensing, the operating system of sensor nodes needs to be reprogrammable to accommodate sudden variations in the environment. In this project, we have decoupled application parameters for Wireless Sensor Networks (WSN) and have developed a library for the LiteOS (a fairly recent WSN operating system) kernel to provide an interface to the user to make applications reactive to the changing environment. We have also implemented a stack sharing mechanism in LiteOS that supports Y-Threads, a combination of events and threads. Current WSN operating systems support either event-driven or thread-based approach which can sometimes be inadequate. The implementation of shared-stack and decoupling application parameters from the kernel on LiteOS shows significant performance improvement over a class of applications that might otherwise over-burden the low-power sensor nodes.

Index Terms— Wireless Sensor Network, Decoupling, Multithreading.

1. INTRODUCTION

Typically, the commonly available sensor-nodes are battery operated and have very limited memory, often only a few kilobytes. While using sensor nodes for such large-scale applications where multiple application threads are involved, the limited memory available within the nodes will be easily overwhelmed. Further, the applications are inherently concurrent, thereby affecting the reliability (predictability) of the system. Thus the operating system must ideally be optimized to support concurrency, save memory and conserve energy.

There are numerous approaches to reduce energy requirements in a wireless sensor network environment. Wang et. al [1] analyze common energy conservation mechanisms at the operating system for wireless-sensor network for large-scale deployments with harsh environmental conditions. The ideal way to reduce energy requirements is to program the application to activate the sensor less frequently. In case of some abnormal behavior, it is important to make the sensor operate at a higher frequency so that we do not miss this event. Current research does not suggest any method of decoupling the task parameters from the application and the only way out would be to redeploy the application with the new parameters. This would mean that a task at the sensor network's base station (sink) has to continuously monitor the sensed data and react in case there is an event of elevated interest. We have tried to come up with a solution to this problem. We provide an interface to application programmers for changing application parameters seamlessly without having to redeploy. This covers our first goal for this project.

The other goal of our project is to optimize memory usage in multithreaded operating systems for the highly resource constrained sensor motes. To support concurrency, WSN operating systems implement either a multithreading model or an event driven model or a restricted combination of both. LiteOS [2], a fairly recent WSN operating system, provides UNIX like abstraction and supports multithreading but does not use any kind of stack optimization techniques. In LiteOS, a private stack is allocated for each thread assuming the worst case execution behavior of that thread. But not all threads use the amount allocated and even if they use, they might not use it at the same time. In this project we have implemented a shared stack mechanism for optimizing stack usage in LiteOS. In particular, we have implemented Y-threads [3] that support separate small stacks for the blocking portions of each thread and a common shared stack for the non-blocking computations.

2. APPLICATION DECOUPLING

We envision that LiteOS should be flexible to allow the application to set task parameters during execution time without the need to reload the application with respect to the environmental change. By permitting the application to modify the kernel, the sensor nodes can be programmed to switch to a different operating mode. This is accomplished by providing a system call interface to the LiteOS kernel to accommodate application demands.

Further, the detection of an event of elevated interest can be sensed by the application by comparing the data with the threshold set in the application itself. When the criterion is

met, the application will make a system call to the operating system asking it to change one or more of its task attributes. The operating system will address this call by changing the task attributes. For example, on the event of a system call, the operating system may change the priority of a task. This way, the communication overhead of the network is reduced as the application need not be re-ported and it becomes smart enough to sense and redefine itself.

2.1. Motivation for application decoupling on LiteOS

Depending upon the characteristics of the application, the following performance gains are obtained when we implement application decoupling on LiteOS.

The latency between the event of sensing and communicating the data to the sink, the sink re-deploying the application and the sensor to start sensing with the new attributes is very high. In order to reduce the latency significantly, the sensing event is triggered frequently when there is an event of high interest, and not when the environment is normal and static.

This dynamic approach is implemented by decoupling the application and its parameters. The current WSN operating systems do not provide enough decoupling between the application and its parameters. Therefore, the users do not have enough control over the application. Using a set of special interfaces at the kernel level of the operating system, the user is provided the feature to write a simple extension to the application which can modify its behavior on the occurrence of certain special events.

The original scheme is quite energy inefficient because of repeated triggering of the sensor units even when there is no variation in the data that is collected. Apart from that, transmission of sensed data to the base station draws up a lot of energy from the resource constrained nodes. Here, the application is made smart, aiming to reduce energy consumption by not sensing frequently when the environment is stable.

2.2. Implementation of parameter decoupling

In order to implement our parameter decoupling and intelligent sensing ideas, we explored the code base of LiteOS and implementation of the LiteOS operating system on micaz motes. Also, in order to compare it with other advanced sensor network operating system, we compared the implementation of LiteOS with Nano-Rk [4] and TinyOS [5]. Even though these operating systems have similar goals, both these operating systems are quite different from LiteOS's implementation in terms of thread handling and scheduling. Hence we consider them to be good comparison platforms for our project.

In LiteOS, the main aim seems to be focused on programmability and usability from the perspective of the developers and users of the system. To this end, the developers of LiteOS have added a lot of features to allow programmers to program the motes and visualize how their application is performing. They have added UI aids for deploying and monitoring applications. The debugging support provided is also very nice. One of their aims seems to make their operating system compatible even for the smallest motes available. To this end, they have tried to keep the code footprint as minimal as possible. An application running on LiteOS can simply start a thread and call one of the sensing functions implemented in the LiteOS library and then start transmitting the same to the base station. The entire operation is executed in a while(1) loop with hard-coded intervals for which the thread is expected to sleep. The code snippet below would explain the same.

```
while (1) {
        reading = get_light();
        radioSend(1, 0xffff, 2, &reading);
        sleepThread(125);
}
```

As we can visualize, a fixed sleep interval is placed after every sensing activity. One of the aims of our exercise is to decouple this parameter from the application. Also, the LiteOS scheduler is very simple and does not ensure any timing guarantees on the task at hand. For example, the priority of the task is set by the scheduler and not by the application. We would like to abstract this functionality so that tasks can run with elevated priority as and when needed. To achieve this aim, we would like to extend or add a new library class to LiteOS which would have the interface to allow applications to make system calls and change parameters which are otherwise set by default in the kernel. (We are at present developing this library and designing an application to demonstrate.)

In comparison to LiteOS, Nano-Rk has a much larger code footprint; applications also need to set several scheduling parameters which makes them highly decoupled but complex at the same time. The scheduler is also much advanced compared to LiteOS and schedules applications using rate monotonic scheduling which is priority based. The scheduler is responsible for maintaining timing guarantees as set through the application. The snippet below shows a typical Nano-Rk task instance.

```
void nrk_create_taskset() {
        nrk_task_set_entry_function( &TaskOne, Task1);
        nrk_task_set_stk(&TaskOne,Stack1,
                        NRK_APP_STACKSIZE);
        TaskOne.prio = 1;
        TaskOne.FirstActivation = TRUE;
        TaskOne.Type = BASIC_TASK;
        TaskOne.SchType = PREEMPTIVE;
        TaskOne.period.secs = 0;
        TaskOne.period.nano_secs = 250;
        TaskOne.cpu_reserve.secs = 1;
        TaskOne.cpu_reserve.nano_secs=50*
```

```
                                        NANOS_PER_MS;
        TaskOne.offset.secs = 0;
        TaskOne.offset.nano_secs= 0;
        nrk_activate_task (&TaskOne);
}
```

So, as we can see, parameters are fixed and hardcoded in the application again. Once this task is submitted to the kernel, it is scheduled to meet all the timing requirements that are set. So, the programmer has to redeploy the task in order to change any of these attributes. Similar to LiteOS, a new library that allows applications to make system calls to the kernel and change some of the attributes that are set here would be beneficial from the programmer's point of view. The task would now be able to sense an interesting event and direct the kernel to change some of its parameters such as the cpu_reserve or the period.

2.2.1. Decoupling Library

We have implemented a library in LiteOS called *appControl*. This library provides an interface to application developers to make system calls to the kernel and elevate the priority of the task at hand. The application developer can now choose from a number of functions. The current supported tasks are listed below:

Increase the priority of the task to make it the task running at highest priority. To implement this, the kernel basically iterates over all the threads running and assigns a priority which is a unit value greater than the highest priority task running currently on the kernel. [Function: *ElevatePrioSetMaxPriority*]

```
while (1)
 {
        reading = get_light();
        yellowToggle();
        if (reading < 10 && state == 0)
        {
                ElevatePrioSetMaxPriority();
                sleepThread(500);
                state = 1;
        }else{
                if(state == 1){
                normalizePriority(oldPrio);
                }
               state = 0;
        sleepThread(1000);
        }
 }
```

The above code shows two system calls: *ElevatePrioSetMaxPriority* and *normalizePriority* which are a part of our library. The implementation of these functions is shown below:

```
void ElevatePrio_setMaxPriority(){
        int credits = 0;
        int i;
        for (i = 0; i < LITE_MAX_THREADS; i++)
   {
      if (thread_table[i].state == STATE_ACTIVE)
      {
         if (credits < thread_table[i].priority)
         {
            credits = thread_table[i].priority;
         }
      }
   }
        credits = credits + 1;
        if(current_thread->priority >= credits ||
current_thread->remaincredits >= credits){
                // do nothing
        }else {
        current_thread->priority = credits;
        current_thread->remaincredits = credits;
        }
}
```

Allow the programmer to increase the priority by a given amount. The CPU time is shared between tasks in proportion to the priorities. For example if there are three tasks with priorities P1, P2 and P3, the CPU time will be shared as P1:P2:P3. Now if the programmer wants to increase the priority P1 by an amount x, the tasks will share CPU time in the ratio P1+ x : P2 : P3. Hence, the task with Priority P1 will now enjoy a larger ratio of the CPU time compared to the other threads. [Function: *elevatePrioSetUserPriority*]

Allow the programmer to boost the task for one single time. This will make the task which requires priority be boosted for a very short time. Internally, a system call increases the *remainingcredits* variable by an amount requested by the application. So the task will enjoy higher priority till its credits are consumed. The application developer has the freedom to increase the remaining credits by an amount as desired by him once or he can keep on increasing as long as the interesting event is taking place. (Multiple calls can be prevented by keeping a flag which checks if a system call has already been made. This is shown in the code segment below). [Function: *elevatePrioSetRemCredits*]

```
if (reading < 10 && state == 0) {
                elevatePrioForTimeInterval(sleepTime);
                sleepThread(5000);
                state = 1;
}
else{
                state = 0;
                sleepThread(1000);
 }
```

Allow the programmers to increase the frequency which allows a task to run with lower sleep time. This can be easily achieved in the application itself by providing a small condition that changes the sleep time of the thread. No separate modifications in the kernel and library are required for this.

Allow a thread the privilege to set all other threads to sleep while it keeps running. This is achieved through a system call where the current thread makes a system call and asks the kernel to put all the other threads to sleep for a required interval of time and leave itself with the entire CPU time. This function should only be used in critical cases where you want only your thread to run on the CPU. [Function: *elevatePrioForTimeInterval*]

Like priority elevation, mechanisms should be there to decrease the priority to normal once the critical period is past. To do this, another system call is invoked which diminishes the increased priority of the task to its old value. [Function: *normalizePriority*]

Continuing effort: Looking more on the lines of flexibility, some scheduling algorithms can work better than others and provisions to change the scheduling algorithm with a system call may prove to be a reasonable and efficient scheme for several applications.

2.3. Problems

The priority elevation through system calls and intelligent applications may prove to be a great help for application developers and maintainers as a lot of energy is saved as tasks need not be re-deployed on the motes. This saves a lot of energy. But while experimenting we came across various bugs which if encountered could defeat the purpose of the decoupling library. Some of the bugs are listed below.

2.4.1. Concurrent requests

One of the uncertainties of the system is what would happen if two threads want extra CPU resources at the same time? This would potentially lead to a race condition and if the first thread puts the other threads to sleep, they may miss an interesting event of their own. The only solution to this is that application developers should use elevate priority methods in case they have potential threads that may also call the same function at the same time. They should put other threads to sleep only in case of some critical applications.

2.4.2. Stack overflow

Another problem is the problem with stack overflow. A user thread may be initially allotted a small amount of stack but once the thread starts making system calls, the stack requirement increases as all these called functions execute on the thread stack. Hence threads that have the potential to make a lot of system calls should be allotted higher amounts of stack. Another method to tackle this problem is the use of run to completion routines discussed in the next section. Also, instead of making continuous system calls flags should be used.

2.4. Extensions and future work

Similar to LiteOS, the ideas of decoupling and priority elevation can be applied to other operating systems. In this project we could evaluate only one other operating system which is Nano-RK. Nano-RK is similar in terms of LiteOS in terms that it offers fixed priority scheduling but unlike LiteOS, allows the user to set a lot of parameters at the application level. These parameters include the task's period (i.e. frequency with which the task should operate), cpu_reserve (i.e. the amount of CPU time that need to be reserved for the task), as well as the scheduling type which is preemptive or non-preemptive. Once a task is deployed, these parameters remain associated with the task till the task is killed and redeployed. Similar to the library we have built for LiteOS, a library can be built for Nano-RK that can make system calls to change these parameters for the task.

When a task in Nano-Rk is executing, it can retrieve its own task Id (The function nrk_get_pid() performs this lookup where it retrieves the id through nrk_cur_task_TCB->task_ID). Once the task id is retrieved, the remaining parameters can be easily set by changing attributes of the task object in the nrk_task_TCB datastructure. [eg. nrk_task_TCB[task_ID].cpu_remaining= new_cpu_remaining].

Since, we were able to extend our idea to one other popular operating system, we expect that we can extend it to other operating systems as well.

Another interesting extension is to provide a two way communication in the system. At the time applications make system calls asking for priority elevation, we cannot guarantee that all requests would be honored. The kernel will decide at the runtime whether the application can or cannot increase the priority of its thread. As a further enhancement, the node in concern here could alert neighboring nodes to start a similar activity or change their attributes in a similar fashion. Though this could be a very interesting act, there could be serious security issues involved as hackers could hack and application and turn the whole network awry.

2.5. Evaluation

Evaluating the decoupling scheme was one of the tough jobs in this project. The library indeed helps to fulfill the application redeploying problem and makes transitions much easier and smoother but on the other hand, the application developer now has to foresee several scenarios and make the application respondent to each of them. The complexity of the applications is definitely increased. The code footprint is also larger but by a very narrow amount (34.4% has become 35.1% after the addition of system call priority elevation support). The memory footprint is an important index since LiteOS boasts of its portability to the smallest nodes because of its small memory requirement.

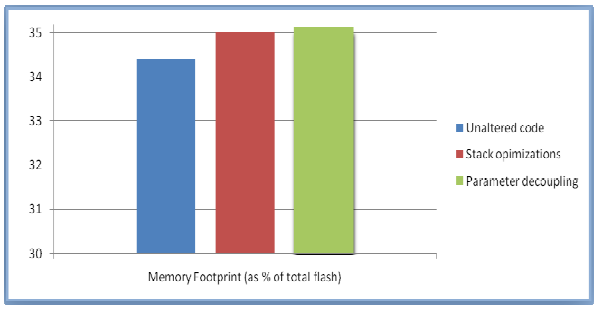

Figure 1: Memory footprint as a percentage of total flash memory after implementing application parameter decoupling and stack optimizations using Y-Threads.

We also measured the time required for porting the kernel on the motes with and without the changes, and they are the same. Hence, we can safely say that the additional code does not cause any significant overhead. One of the major improvements that application decoupling offers is that it removes the latency for re-deploying an application. The only cost which the system pays is a single system call.

3. Y-THREADS

Nitta et. al. [3] proposes Y-Threads that separates the control and computational portion of each thread and uses a separate stack for each portion. Y-Threads are preemptive multithreads that run in a small private stack and all the non-blocking routines are executed in a separate common stack. Thus Y-Threads support preemptive multithreading and at the same time make efficient use of the limited RAM.

The observation behind using Y-Threads is that the control portions of applications require a small amount of stack which can be pre-allocated to each thread. To execute the computational portions of the threads the kernel provides a separate shared stack. Thus Y-Threads are just like the normal threads with the exception that the majority of work in Y-Threads is done by Run to Completion Routines (RCR) that are executed in the shared stack. Consider the following example:

```
/*Normal Implementation*/

while(1) {
        recv_radio_msg(&msg);
        leds_greenToggle();
        processMsg(&msg);
}

/*RCR Implementation*/

while(1) {
        recv_radio_msg(&msg);
        leds_greenToggle();
        rcr_call(processMsg, &msg);
}
```

Here a thread accepts a message and then calls a routine processMsg() to process the message. This processMsg() routine involves some computation and requires memory. In the Y-Thread based stack sharing mechanism this routine will be executed as a RCR routine and will be executed in a shared stack (Figure 2). The shared stack is allocated by the kernel and is used to execute all such routines from different threads.

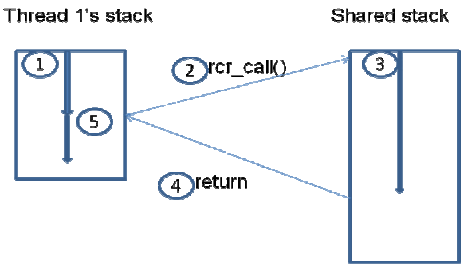

Figure 2: Step1: Thread 1 is executing in its own stack, Step 2: Thread 1 makes an rcr_call, Step 3: Execution is started in the shared stack, Step 4: rcr_call() returns, Step 5: Execution is resumed in thread's own stack. The arrow indicates the execution of a thread in the stack.

3.1. Implementation

We have implemented Y-Threads in LiteOS. We have modified the operating system kernel and the thread library and verified our system by running it in MicaZ wireless sensor motes with AVR processors. The MicaZ motes have only 4096 bytes of RAM among which the basic LiteOS kernel (before our modifications) occupies 2562 bytes [6]. The kernel stack grows from the highest RAM address and it is recommended to reserve addresses higher than 3650 for the kernel stack. Thus applications actually have approximately 1K RAM available for all purposes. Since the amount of RAM is very limited it is really necessary to

optimize RAM usage in these motes and this is exactly what Y-Threads do.

To support Y-Threads the operating system needs to have support for two basic functions; one for creating the Y-Threads and one for executing the run to completion routines; the functions are specified below:

```
/*Interface to create threads with their own private stacks*/

typedef void (*y_thread_task) (void);
void createThread(y_thread_task task, uint16_t *ram_start, uint16_t *stack_pointer, uint16_t priority, char *thread_name)

/*Interface to execute run to completion routines in the shared stack*/

typedef void (*rcrFunction) (void *rcrData);
void rcrCall (rcrFunction func, void *rcrData)
```

Since Y-Threads are created just like normal threads, the createThread() method in LiteOS was completely suitable for creating Y-Threads as well. To support the RCRs we have modified the threads library in LiteOS and added the new function rcrCall(). rcrCall() takes as arguments the pointer of the function to be executed in the shared stack and a pointer to the data to be passed to that function and passes these parameters to the kernel through registers. The system call corresponding to this library function is implemented in the kernel; the system call retrieves the parameters from the registers, switches the stack pointer (__SP__) register so that __SP__ register points to the shared stack and then executes the function pointed to by the retrieved function pointer. Almost all of the codes in both the library function and the system call are written in inline assembly for the AVR processor. The pseudocode of the system call is as follows:

```
/*System call to execute a function in the shared stack*/

Function  rcr_call()
Begin
        Start atomic execution
        Retrieve function pointer and data from registers
        SWAP_STACK_POINTER(backupStackPtr, sharedStackPtr)
        Push rcr data onto the shared stack
        Execute ICALL instruction
        Pop rcr data from the shared stack
        SWAP_STACK_POINTER(sharedStackPtr, backupStackPtr)
        End atomic execution
End
```

SWAP_STACK_POINTER is a macro written in inline assembly that saves the value of the current stack pointer register (__SP__) in the first parameter given and loads value of the second parameter into the __SP__ register. Thus after the first call to the SWAP_STACK_POINTER() macro the __SP__ register is backed up in the backupStackPtr and a pointer to the shared stack is loaded into the __SP__ register. The actual thread stack pointer is restored when the RCR returns. The RCRs are assumed to be non-blocking and cannot be preempted. Thus the whole execution in the shared stack is done atomically.

3.2. Observations

The Y-Thread implementation requires the availability of a shared stack to execute the RCRs. The shared stack is actually a chunk of memory reserved by the kernel. There are two possibilities to implement the shared stack:

(1) Maintain a shared stack always, and

(2) Create the shared stack whenever necessary and release the memory when the shared stack is no longer needed.

Nitta et. al. proposed the first approach in [3]. The problem with this approach is that not all applications will require the shared stack; but the kernel will maintain a shared stack always even when the applications do not need it. Wireless sensor motes usually have very small amounts of RAM and maintaining a shared stack when it is not needed results in overkill of limited RAM.

To account for this we have added two more functions to the thread library, one for requesting the initial setup of the shared stack and one for releasing the shared stack when it is no longer needed. In particular we have added the following library functions:

```
/*Interface for initializing the shared stack*/

initSharedStack(uint16_t size)

/*Interface for releasing the shared stack*/

releaseSharedStack()
```

When the kernel receives the first call from any thread requesting the initialization of a shared stack, it allocates the specified amount of memory and maintains it as the shared stack. The kernel maintains a counter that counts how many threads are using the shared stack and the counter is increased during each call to initSharedStack() and is decreased during each call to releaseSharedStack(). When all the threads release the shared stack the kernel frees up the memory that was allocated for the shared stack.

The advantage of this approach is that it is robust and is suited to both Y-Threads and normal threads. If no thread needs a shared stack, no memory will be allocated, and there will be almost no overhead (other than the increased OS code footprint) for supporting Y-Threads. The disadvantage of this approach lies in the overhead of two system calls per Y-Thread; one for requesting the shared stack and the other for releasing it. Another problem with this approach is that if

the shared stack initialization request arrives at some point when the memory is already fragmented, the kernel may fail to allocate the chunk required for the stack.
One work around to avoid the extra system calls would be to incorporate this information in the thread creation and destruction system calls. For example, the createThread() function can be easily extended to accept one more parameter specifying whether it needs a shared stack or not. This functionality is not implemented yet.

3.3. Limitations of Y-Threads

The most important limitation of Y-Threads is that it is not suitable for all types of applications. Y-Threads can only reduce RAM usage when the threads execute some compute intensive tasks that can be moved to the shared stack. Most basic WSN applications just take readings from some sensors such as temperature or light sensors and send the readings back to the base station where the readings are actually processed. Y-Threads can add no benefit to these applications. However, the wireless sensor motes are becoming more powerful these days and many compute intensive applications are being developed for the motes which can be benefitted from Y-Threads. An important task for all sensor networks is time synchronization [7] which involves floating point computation and memory overhead. This task can easily be ported to the shared stack. Sensor network applications also perform Fourier Transformation (in motion/event detection), Cyclic Redundancy Checks to ensure reliability in health/military services and compression/decompression of images to reduce energy consumption in data transfer. All of these applications will be benefitted from the Y-Thread based shared stack approach.
Another limitation of Y-Threads lies in the overhead of extra system calls. Since no thread can be preempted while being executed in the shared stack; the tasks that are executed in the shared stack should be small. Each execution request of these tasks incurs a system call overhead which results in more CPU cycles and additional energy consumption.

3.4. Evaluation

To verify the correctness of our implementation we have used the benchmarks that come with the LiteOS distribution. We modified the applications by wrapping up the regular function calls with rcr_calls and verified that the applications behave as expected.
However, these applications actually do not involve any processing that requires memory and hence are not suitable to evaluate the benefit of shared stack. Unfortunately, memory intensive WSN benchmarks (references can be found in [8]) are not available. So, we were not able to run any experiment regarding this. To provide an estimation of how much memory can be saved from shared shack we have analyzed the Fourier Transform (FT) application from the MiBench benchmark suite. We have found that the fourierTransform() function uses more than 90 bytes of stack only for the local variables and the nested function calls. The basic stack requirement of any thread in the MicaZ motes is around 50 bytes (including stack space for saving 32 registers on a context switch). Thus we can allocate two more threads by executing the FT function in the shared stack. Obviously, the shared stack needs to allocate the memory, but that memory will be allocated only once and will be used by multiple such threads.

3.5. Future work

In the currently implemented version of Y-Threads only run to completion routines are executed in the shared stack. An extension of this project is to implement the RCRs as non blocking routines (NBR). When implemented as NBRs Y-Threads remain preemptable even when running in the shared stack; the only restriction is that the routines cannot block while running in the shared stack. To implement NBRs the current top of the NBR stack must be stored during a context switch.
Another extension is to implement a completely shared stack for all threads. In a completely shared stack no thread will be allocated with a private stack; instead all threads will run on the same shared stack implemented by the kernel. A thread's stack will be eventually fragmented due to preemption and the kernel will have the responsibility to link the chunks that are used by that thread. For each thread, a doubly linked list can be used to identify its chunks in the shared stack. This extension requires support from the hardware; it can also be done using compiler support by wrapping up the Push and Pop operations. But in that case a table lookup and an if-else logic need to be incorporated with each push and pop operation which will incur a lot of CPU overhead.

4. CONCLUSION

In this project we have provided library and kernel support to decouple application parameters in wireless sensor network applications. Decoupling enables the application developers to make their applications adaptable to the changing environment. It saves a lot of energy and time; since applications can now change their attributes on the fly and need not be re-deployed. We have successfully added the decoupling support in LiteOS, a UNIX based operating system for WSN.
We have also implemented a shared stack based memory optimization technique (Y-Thread) in LiteOS. The Y-Thread approach allocates a small amount of dedicated stack to each thread and enables the execution of any run to completion routine in a kernel provided shared stack. By re-using the

shared stack, the Y-Threads based approach makes efficient utilization of the limited RAM in wireless sensor nodes.

REFERENCES

[1] Lan Wang and Yang Xiao, "Energy Saving Mechanisms in Sensor Networks", in the ACM Journal on Mobile Networks and Applications, Vol 11:5, Oct 2006.

[2] Cao, Q., Abdelzaher, T., Stankovic, J. and He, T. The LiteOS Operating System: Towards Unix-like Abstractions for Wireless Sensor Networks, In the Proceedings of the 7th international conference on Information processing in sensor networks, 2008.

[3] Nitta, C. and Pandey, R. and Ramin, Y. Y -Threads: supporting concurrency in wireless sensor networks, In Proceedings of the International Conference on Distributed Computing in Sensor Systems, 2006.

[4] Eswaran A. Rowe, A. and Rajkumar. R. Nano-RK: an Energy-aware Resource-centric RTOS for Sensor Networks, In the Proceedings of the 26th IEEE International Real-Time Systems Symposium, 2005.

[5] Hill, J., Szewczyk, R., Woo, A, Hollar, S., Culler, D., and Pister, K. System architecture directions for network sensors. In Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), Cambridge, MA, 2000IEEE Signal Processing Magazine, Vol. 25(2), March, 2008.

[6] LiteOS Programmers' Guide, http://www.liteos.net/docs/LiteOS_Programming_Guide.pdf

[7] M. Maroti, B. Kusy, G. Simon, A. Ledeczi, The Flooding Time Synchronization Protocol, Proceedings of the second international conference on Embedded networked sensor systems, 2004.sensor systems, 2004.

[8] Gurhan Kucuk and Can Basaran. Reducing Energy Consumption of Wireless Sensor Networks through Processor Optimizations. Journal of Computers, Vol. 2, No. 5, July 2007.

[9] Han, C. C., Rengaswamy, R. K., Shea, R., Kohler, E., and Srivastava, M. SOS: A dynamic operating system for sensor networks. In Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services (Mobisys), Seattle, WA, 2005.

[10] Duffy, C., Roedig, U., Herbert, J. and Sreenan, C. An Experimental Comparison of Event Driven and Multi-Threaded Sensor Node Operating Systems, In Proceedings of the Fifth IEEE International Conference on Pervasive Computing and Communications Workshops, Washington, DC, USA, 2007.

[11] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In proceedings of Emnets-I, 2004.

[12] Zhou, H. and Hou K. LIMOS: a Lightweight Multi-threading Operating System dedicated to Wireless Sensor Networks, In Proceedings of the International Conference on Wireless Communications, Networking and Mobile Computing, 2007.

[13] Krishna Chintalapudi, Tat Fu, Jeongyeup Paek, Nupur Kothari, Sumit Rangwala, John Caffrey, Ramesh Govindan, Erik Johnson, and Sami Masri, "Monitoring Civil Structures with a Wireless Sensor Network", IEEE Journal on Internet Computing, March-April 2006.