Y-Threads: Supporting Concurrency in Wireless Sensor Networks

Christopher Nitta, Raju Pandey, and Yann Ramin

Department of Computer Science
University of California, Davis
Davis, CA 95616
{nitta, pandey, ramin}@cs.ucdavis.edu

Abstract. Resource constrained systems often are programmed using an event-based model. Many applications do not lend themselves well to an event-based approach, but preemptive multithreading pre-allocates resources that cannot be used even while not in use by the owning thread. In this paper, we propose a hybrid approach called Y-Threads. Y-Threads provide separate small stacks for blocking portions of applications, while allowing for shared stacks for non-blocking computations. We have implemented Y-Threads on Mica and Telos wireless sensor network platforms. The results show that Y-Threads provide a preemptive multithreaded programming model with resource utilization closer to an event-based approach. In addition, relatively large memory buffers can be allocated for temporary use with less overhead than conventional dynamic memory allocation methods.

Keywords: Stack sharing, Multi-threading, Concurrency.

1 Introduction

Wireless Sensor Network (WSN) systems are inherently concurrent. Support for concurrency is needed in all layers of the WSN software stack. At the operating system level, hardware interrupts, low level I/O and sensor events are asynchronous. At the middleware level, specific services (such as time synchronization [1][2] and code distribution[3][4][5]) are highly concurrent in nature, and exist independently from other activities. For instance, most code distribution protocols have several concurrent activities: one may actively maintain a code distribution tree by periodically collecting neighbor information, while the other may cache and distribute code along the distribution tree. At the application level, programs may define both node-level (e.g., collect data) and group level activities (e.g., aggregate data), each occurring concurrently.

Concurrency exists not only at many different levels, but also in many different forms: rapid responses to specific events are easily represented using events and event handlers; concurrency among middleware services are better expressed using long running threads (LRT); concurrency and group concurrent activities are better defined using a combination of threads and atomic computations; and higher level system software abstractions (such as virtual machines and middleware) can be implemented

P. Gibbons et al. (Eds.): DCOSS 2006, LNCS 4026, pp. 169 – 184, 2006.

easier using a threading mechanism. We have examined these concurrency models and believe there is a disconnect between the two main models.

WSN Operating Systems work with limited resources, RAM being the primary limitation. The limited RAM drives many embedded system designers to use an event-based programming model as in TinyOS[6] and SOS[7]. Though [8] shows that event-based and thread-based approaches can be interchanged, many applications do not lend themselves well to an event-based approach, especially those where true CPU concurrency is needed[9][10]. Often embedded system tasks run a cycle of work and waiting. Blocking is done at the highest level, computation is executed to completion and waiting occurs again. Preemptive multithreading pre-allocates RAM that cannot be used even while not in use by the owning thread. What is needed is a concurrency model that balances the programming needs through a preemptive threading model and at the same time meets the resource constraints of sensing devices.

We introduce a hybrid approach called Y-Threads. Y-Threads are preemptive multithreads with small thread stacks. The majority of work in Y-Threads is done by non-blocking routines that execute on a separate common stack. By separating the execution stacks of control and computational behavior, Y-Threads can support preemptive threading model with better memory utilization than preemptive multithreading alone.

We have implemented Y-Threads on several WSN platforms. Experimental results show that a Y-Thread version of a time synchronization application only increased energy consumption by 0.12% over the original purely multithreaded version. In addition the worst-case RAM requirement for the Y-Thread implementation was reduced by 16.5%. Experiments also show that Y-Thread implementations of a flash modification routine are more processing efficient than versions that dynamically allocate memory.

The rest of this paper is structured as follows. Section 2 discusses the motivation, programming model, and implementation of Y-Threads. Experimental test applications and results are described in Section 3. Section 4 discusses the existing concurrency models more in depth. Section 5 discusses the possibilities of future work on Y-Threads. We conclude in Section 6.

2 Y-Threads

Y-Threads are preemptive threads and are well suited to capture the reactive nature of many WSN programs. It is based on the insight that many WSN applications block, waiting for specific events to occur, that has motivated the development of Y-Threads. As events occur, they react by performing atomic computations, changing their state, and returning to the wait mode. Behavior of many such applications can be captured in terms of two sets of behavior: the first is a control behavior that is state-based and that guides the application through different state transitions as different events occur. The second are the different computational behaviors that occur during various state transitions.

For instance, consider the time synchronization code sample shown in **Fig. 1**. The control behavior is defined in the while loop: the application blocks while waiting for

events (such as message arrival). Upon occurrences of these events, it performs specific actions (such as *processMsg*), and then goes back to wait for other events to occur.

We observe that the size of stack required to execute the control behavior is fairly small. By separating the execution stacks of control and computational behavior, we can support preemptive threading model and save on memory space. Y-threads implement this idea by providing support for both control and computational behaviors.

```
void timesync_ReceiveTask(){
  TimeSyncMsg msg;
  while(1) {
    recv_radio_msg(&msg);
    leds_greenToggle();
    if((state & STATE_PROCESSING) == 0 ){
      mess.arrivalTime = hal_sys_time_getTime32();
      processMsg(&msg);
} } }

void processMsg(TimeSyncMsg *msg){
  …
}
```

Fig. 1. Time Synchronization code sample

2.1 Y-Threads Programming Model

Y-Threads provide the capabilities of preemptive multithreading with good utilization of limited RAM. Y-Threads are preemptive threads with small pre-allocated stacks. Y-Threads have the same semantics as general threads, but the majority of work is done by Run to Completion Routines (RCR) that execute on a separate common stack. Task stack sharing and a scheduling method for correct system operation are discussed in [11]. The difference in Y-Threads is that RCRs execute as the invoking thread and maintains the invoking threads priority, whereas [11] discusses run to completion flyweight tasks. The Y-Thread interface provides two primary functions, one to create the Y-Threads, and one to invoke the RCRs. **Fig. 2** shows the Y-Thread and RCR APIs. The APIs are discussed in further detail in Section 2.2.

```
typedef void (*oss_task_function_t)(void);
typedef void *oss_rcr_data_t;
typedef void (*oss_rcr_function_t)(oss_rcr_data_t);
oss_taskp_t oss_ythread_create(oss_task_function_t
task_func, uint16_t sz);
void oss_rcr_call(oss_rcr_function_t func,
oss_rcr_data_t data);
```

Fig. 2. Y-Thread and RCR API

Fig. 3 shows the memory map of two different threads each invoking an RCR. Task 1 has higher priority than Task 2 and waits until the semaphore is signaled. The arrow in each portion of **Fig. 3** signifies the current execution stack. In **a** of the figure Task1 is blocked, and Task2 is running. Task 2 invokes an RCR in **b,** which executes on a separate stack and then returns in **c**. Task1 is of higher priority, and is unblocked in **d** and invokes its RCR in **e**. The RCR returns and Task1 continues to execute on its stack in **f**. Notice the sharing of the common stack space for both RCRs. The stack sharing provides programming semantics of a much larger virtual stack for each task. The source in **Fig. 4** is an example that corresponds to the memory map in **Fig. 3**.

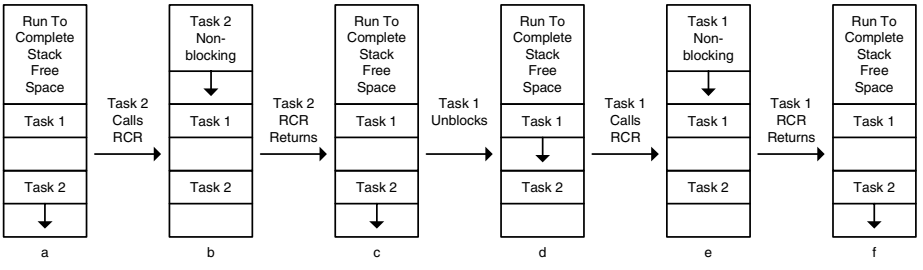

Fig. 3. Y-Threads Memory Usage

```
void Task1(){
  while(1){
    wait_sem(&sem1);
    oss_rcr_call(Task1Process,NULL);
} }

void Task2(){
  while(1){
    oss_sleep(100);
    oss_rcr_call(Task2Proccess,NULL);
    signal_sem(&sem1);
} }

void Task1Process(oss_rcr_data_t *data){
  ...//Do something
}

void Task2Process(oss_rcr_data_t *data){
  ...//Do something else
}
```

Fig. 4. Y-Thread Code Example

Y-Threads are similar to preemptive multithreads that spawn run to completion threads without the overhead of creating a run to completion thread. It may be possible to implement Y-Threads such that an RCR can be preempted by threads of higher priority since threads that are on the RCR stack cannot block. Currently Y-

Threads have not been implemented in this manner and such an implementation should rename RCR to Non-Blocking Routine (NBR).

Parameters can be passed to RCRs and results can be returned, since it is just a matter of copying data to and from the correct contexts. Since the invoking thread transfers execution control to the RCR during invocation, all data on the controlling thread's stack can be accessed by the RCR. The RCR signature is dependent upon the implementation of Y-Threads. The initial implementation of Y-Threads uses a single function to invoke the RCR. RCRs can invoke subroutines as long as none of the subroutines attempt to block execution. If an RCR invokes another RCR, it has the same semantics as invoking a subroutine since the invoking RCR is already executing on the RCR stack.

Y-Thread behavior can be emulated on any RTOS that has both preemptive threads and light weight or run to completion threads with extra overhead. Y-Threads are similar to OSEK[12] Extended tasks that spawn Basic tasks, in those systems that execute Basic tasks on a common stack. The exception is that they do not execute as Extended tasks and do not inherit their priority.

Y-Threads have better memory utilization than just pure preemptive multithreading since large amounts of memory can be automatically allocated and freed on the RCR stack. Preemptive multithreads must pre-allocate enough memory for the worst case stack utilization; therefore there is memory that is unused in each thread, but is also unusable by other threads. Y-Threads do not need to pre-allocate large amounts of memory for potential future use unlike purely preemptive multithreading. Y-Threads can also automatically allocate large amounts of memory on the RCR stack with less overhead than dynamic memory allocation. Y-Threads provide the advantages of preemptive multithreading without the disadvantage of high memory overhead.

2.2 Y-Thread Implementation

Y-Threads were first implemented on OS*[1] for the AVR ATMega128 using an invocation to a helper function *oss_rcr_call*. The signature of the RCRs is a single void pointer parameter with a void return type. The RCR data types and oss_rcr_call prototype can be seen in **Fig. 2**. Light Weight Threads (LWT) are implemented in OS* using an RCR wrapper. The calling convention of the ATMega128 under GCC passes most function parameters in registers making overhead of the *oss_rcr_call* helper function very low. The advantage of compiler support or *oss_rcr_call* in-lining is not likely to be as great for the ATMega128 as it may be for other architectures where parameters are primarily passed on the stack. Switching the control to the RCR stack was relatively straight forward in the ATMega128 since GCC uses registers and the frame pointer to access local variables and parameters.

OS knowledge is necessary for any Y-Thread implementation. OS knowledge is necessary since it must be known if the RCR stack is in use or not. Furthermore, if RCRs are implemented as "Non-Blocking Routines" as discussed in Section 2.1, the current top of the NBR stack must be stored during a context switch from a thread in the NBR. **Fig. 5** illustrates a higher priority thread preempting an NBR. In part a of the figure Task 2 of lower priority is executing a NBR. Task1 is unblocked by an ISR

[1] OS* is a light-weight synthesizable operating system for the Mica, Telos and Stargate families of sensor nodes currently under development by the SENSES group at UC Davis.

in part b and context switch from Task2's NBR to Task1 occurs. Task1 invokes an NBR in part c which executes on the common stack under Task2's ready NBR. Task 1's NBR would overwrite Task 2's NBR if the NBR stack top is not stored during the context switch. Due to the nature of the OS* scheduler, our current Y-Thread implementation does not need to store the top of the RCR stack.

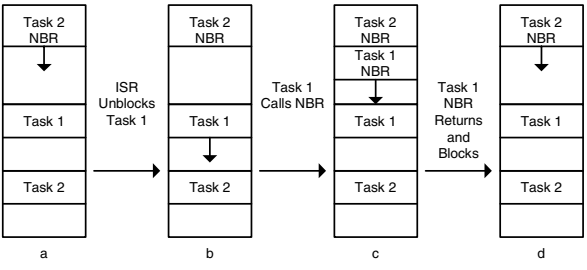

Fig. 5. NBR Preemption by higher priority thread

We also implemented Y-Threads for the TI MSP430. The RCR API is identical to that of the ATMega128 implementation. Unlike the ATMega128, the implementation of Y-Threads for MSP430 was complicated by GCC's use of the frame pointer. The difficulty arose in implementing a version of Y-Threads that operated properly under all optimization levels of GCC. One possible solution is to have implementations for different compiler options; this is similar to libraries that are developed for both banked and non-banked memory models.

3 Applications and Performance Evaluation

Since Y-Threads are preemptive multithreads with small stacks, any application that can be implemented using preemptive multithreads can also be implemented using Y-Threads. The applications evaluated in this paper show the advantage of a preemptive multithreading programming model, and the need for relatively large temporary memory allocations. All applications evaluated were compiled for the AVR ATMega128. All test data was either collected from AVRORA[2] or backed out of object dumps of the applications.

3.1 Run to Completion Routine vs. Subroutine and Light Weight Thread Invocation

The overhead associated with invoking an RCR compared to a normal subroutine is an important metric to evaluate. LWTs are run to completion threads on OS* and therefore are a possible alternative to RCRs for the execution of non-blocking code. A test program was written to evaluate the overhead associated with RCR, subroutine and LWT invocations. **Fig. 6** shows the overhead in instruction cycles for invoking

[2] AVRORA is an AVR simulator developed by UCLA Compilers Group and available at http://compilers.cs.ucla.edu/avrora/

each type of routine. LWTs must be posted to the scheduler prior to the thread switch; therefore the LWT thread switch is shown with and without the scheduler posting overhead. As expected the invocation overhead of the RCR is higher than the normal subroutine, but it is less than half as much as the LWT. The overhead of the RCR (177 instructions) is closer to that of a subroutine (94 instructions) than that of the LWT (462 instructions).

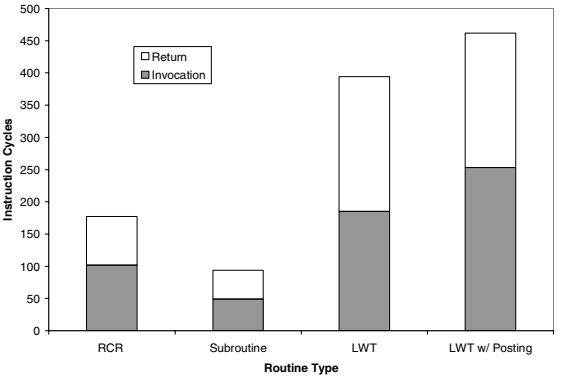

Fig. 6. RCR, Subroutine, and LWT Invocation Times

3.2 Dynamic Allocation vs. Y-Thread Automatic Allocation

Limited resource systems often need to allocate relatively large memory buffers for temporary use. Modifying flash is one such application that is necessary in many embedded systems. The nature of flash requires that entire pages, sometimes as large as 512B in size, be erased before reprogramming can occur. These resource constrained systems typically have between 1KB and 10KB of RAM, and therefore a flash page is relatively large compared to the entire system memory. **Fig. 7** illustrates the copy, erase, modify, write back cycle of modifying a flash page.

There are two methods to allocate relatively large amounts of memory for temporary use in a preemptive multithreading environment. Memory can be dynamically allocated through a function invocation such as *malloc* and then freed when not needed through an invocation of *free*. The other method is to automatically allocate the memory on the stack by invoking a function.

Dynamic memory allocation has the advantage that the allocation size is bound at run-time unlike automatic allocation that is bound at compile-time. The overhead of automatic allocation is constant and is typically lower than dynamic allocation. In a preemptive multithreading environment the drawback to automatic allocation is that the memory must be pre-allocated during the context allocation. The pre-allocated memory is unusable by other threads even when it is not in use by the thread. The advantage of Y-Threads is that they can automatically allocate memory in an RCR with less overhead than dynamic memory allocation. Further when a thread is not in an RCR the memory is available for other threads to use.

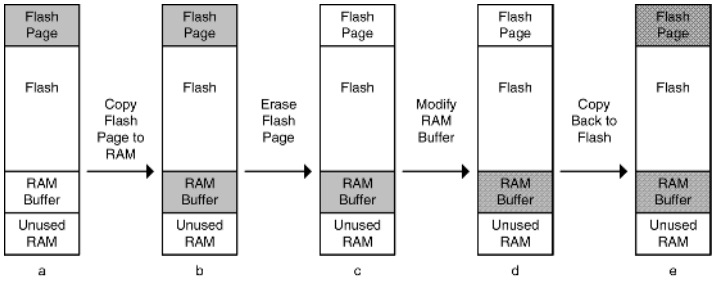

Fig. 7. Flash Page Modification

A flash modification application was implemented using four different methods. All test applications were implemented on OS* for the AVR ATMega128 and analyzed using AVRORA. The first method uses a small thread that dynamically allocates memory for the modification buffer via *malloc*. When the buffer is no longer needed it is freed. The second method utilizes a global buffer for the modification. The final two test applications utilize RCRs that automatically allocate the modification buffer on the stack. One of the methods uses the *oss_rcr_call* while the other inlines the *oss_rcr_call* to emulate compiler support for Y-Threads.

The flash modification function execution time is dependent upon the fragmentation of the heap for the *malloc* version. All other versions of the function were independent of heap fragmentation. **Fig. 8** shows the execution time of each of the function versions normalized to the RCR version. The *malloc* version is slightly slower than the RCR version in the best case and climbs to over 8% higher at only 16 fragments. The RCR inlined version and the global buffer version were 0.7% and 1.7% faster than the RCR version respectively. If Y-Threads were to be supported by the compiler or inlined through optimizations the RCR version would be approximately 1% slower than the global buffer version.

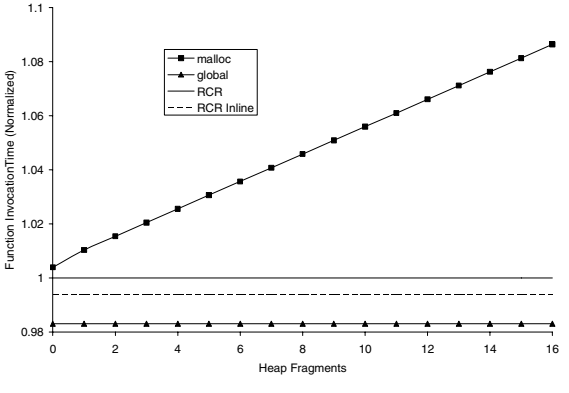

Fig. 8. Flash Modification Time vs. Heap Fragmentation

The *malloc* version while slightly slower than the RCR version in the best case performs worse as memory becomes fragmented. The non-determinism of the *malloc* version makes it undesirable for any application that has real-time requirements. Also the RCR version does not statically allocate memory that cannot be used like the global buffer version does.

3.3 Time Synchronization

Time synchronization is a common service that is often required in WSNs. The time synchronization code illustrated in **Fig. 1** was evaluated for performance. The *processMsg* subroutine was replaced with an RCR version for evaluation. **Fig. 9** shows the code for the RCR version of the application from **Fig. 1**. Notice that very few changes were required. The *processMsg* subroutine has floating point math that is relatively RAM intensive for the 8-bit AVR ATMega128 CPU. The simulation of the time synchronization routine was run on AVRORA for ten seconds.

```
void timesync_ReceiveTask(){
  TimeSyncMsg msg;
  while(1) {
    recv_radio_msg(&msg);
    leds_greenToggle();
    if((state & STATE_PROCESSING) == 0 ){
      mess.arrivalTime = hal_sys_time_getTime32();
      oss_rcr_call(processMsg, & msg);
} } }

void processMsg(oss_rcr_data_t *data){
  TimeSyncMsg *msg = (TimeSyncMsg *)data;
   ...}
```

Fig. 9. RCR Time Synchronization Code Sample

The energy consumption and the number of active CPU cycles were compared for the original and the RCR versions of the time synchronization algorithm. **Fig. 10** shows the results of the tests. The RCR version was active for 0.12% more instruction cycles than the non-RCR version, and consumed 0.02% more energy. The overhead of running the RCR version is insignificant in terms of energy usage.

RAM utilization was also compared for the both the RCR and original versions. **Fig. 11** illustrates the worst-case, thread stack, interrupt and RCR stack memory usage for both the original and RCR versions. The worst-case memory was calculated as the total memory required for maximum function invocations plus the maximum ISR requirements. The worst-case memory usage originally started at 278 bytes and was reduced to 232 bytes in the RCR version. Since the processMsg call was switched to an RCR call, the thread stack requirements were reduced from 142 to 58 bytes. The transition of processMsg to an RCR call alone should actually increase the total required RAM, but when interrupts are considered the worst-cast RAM utilization is actually reduced in the RCR version. In the original version LWTs execute on the

scheduling stack, whereas the RCR version LWTs utilize the RCR stack. Sharing of the RCR stack is the main reason for the reduction in worst-case RAM utilization from that of the original version. As the number of Y-Threads increases, the worst-case RAM savings of RCR versions should increase because more RAM should be shared on the RCR stack.

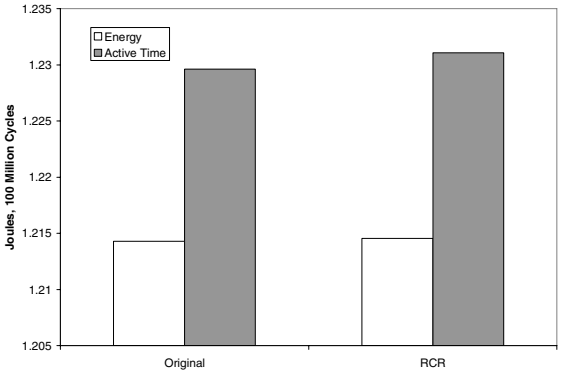

Fig. 10. Time Synchronization Energy and Active Time

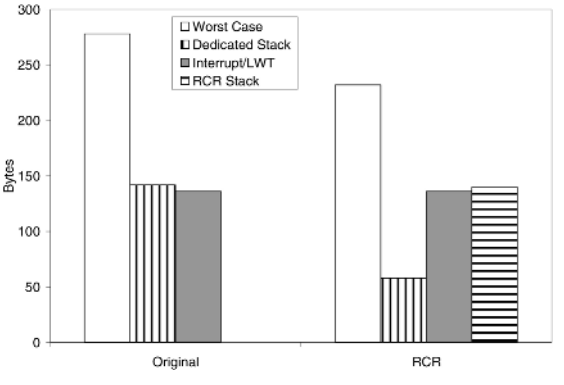

Fig. 11. Time Synchronization Memory Usage

3.4 "Delta" Based Code Distribution

The distribution of a new code image to an embedded system is difficult and often requires a significant amount of system resources. The method discussed in [13] of transmitting only deltas for system updates has much promise and could benefit from the use of Y-Threads. In order to minimize data transmission, deltas or differences between the new and existing image are transmitted.

The update application receives deltas and modifies flash pages. As discussed in [14], implementing state machines is often easier using preemptive multithreading than event based programming. The reception of delta packets can easily be done in a preemptive thread. Since the size of the deltas are relatively small compared to the size of a flash page, the delta packets can be stored on the thread stack.

A large buffer is required to construct the new flash pages from the deltas and the existing image. As discussed previously, the nature of flash requires that the entire page be erased prior to reprogramming. Prior to the implementation of Y-Threads, a statically allocated global buffer was used to implement the delta update application. A RCR can be used to construct the updated flash pages from the deltas and the existing image. The large buffer can be allocated on the RCR stack as discussed in the previous section. The delta update RCR constructs the new flash page in RAM, erases the page to be updated, and then reprograms the page.

4 Concurrency Model Discussion

The two main concurrency models are event-based microthreads, and preemptive multithreads. Event-based microthreads have very low overhead both in processing and memory utilization. Preemptive multithreaded systems allow for per thread state maintenance and thread blocking. The preemptive multithreaded programming model makes application development easier when true CPU concurrency is necessary.

Event-based microthreads and preemptive multithreading are not the only concurrency models that exist. Lazy threads and protothreads are two other pseudo-concurrency models. Both lazy threads and protothreads allow for blocking and are implemented using co-routine like mechanisms. Protothreads have been used successfully in Contiki[14][15]. All of the concurrency models must also deal with interrupts; therefore we discuss various implementation techniques for interrupt handlers.

4.1 Event-Based Microthreads

Event-based programming does not require a separate stack per execution context. Event handlers are typically implemented purely as function invocations making events efficient both in execution and memory utilization. Event-based systems often can be implemented in a high level language making the system easier to implement and more portable. **Fig. 12** shows the single stack utilization of event based threads.

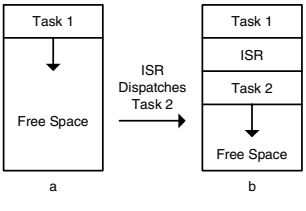

Fig. 12. Event-Based threading stack

The disadvantage of the event-based concurrency model is that events or "microthreads" must be run to completion; events are not allowed to block. The ease of system development can come at the cost of complicated application development for systems that require true CPU concurrency. A further disadvantage of microthreads is that state maintenance across event handler calls must be done via global variables [16]. The use of global variables to maintain state may limit the modularity that can be achieved especially if written in a language such as C.

4.2 Preemptive Multithreading

Preemptive threads require RAM for each execution context. Preemptive threads allow for blocking, and per thread state maintenance. State maintenance with true CPU concurrency is often easier to implement in a multithreaded environment. Often WSN applications need to implement communication protocols with similar timeouts and system states. The increased modularity and ease of application programmability comes at a cost.

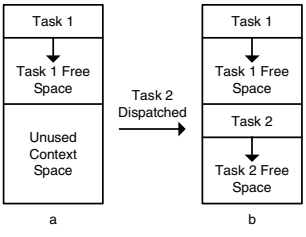

Fig. 13. Preemptive multithreading memory

Preemptive threads require assembly at the systems level for the context switch, complicating systems development. The context switch also has higher overhead than a function invocation. Preemptive threads must pre-allocate enough RAM to execute the deepest sequence of function invocations, and therefore hold an unused resource most of the time. Pre-allocation of memory is illustrated in **Fig. 13**. The pre-allocation of memory is the main drawback that has driven the use of event-based programming model in many embedded systems.

4.3 Lazy Threads

Lazy threads are a threading model that allows for parallel function invocations. The need for fast thread forking has driven the development of lazy threads. Lazy threads implementations attempt to allocate child threads on the parents stack in a "stacklet"[17]. If blocking of the child is necessary an entirely separate context must be allocated for the child and the child thread must be transferred. Unfortunately lazy threads are non-preemptive and therefore require cooperative multithreading. Implementations of lazy threads require compiler support to properly allow for continuations. No pointers to stack data are allowed since there is a possibility of context relocation, further limiting the flexibility of lazy threads.

4.4 Protothreads

Protothreads are pseudo threads that use a coroutine technique to implement blocking threads. Protothreads all execute on a single stack and are implemented entirely in C. The advantage of Protothreads is that they best utilize limited resources while providing a somewhat preemptive multithreading programming model. **Fig. 14** shows the coroutine thread intertwining on a single stack. There are a few limitations of Protothreads; a major limitation is that automatic variables are not saved across blocking waits. The automatic variable limitation is not intuitive, and therefore can lead to programming mistakes. Blocking waits may only occur in the Protothread function and cannot occur in subroutines called by the Protothread. The blocking wait subroutine limitation is not as severe as automatic variable limitation especially when considering that blocking is often done at the highest level.

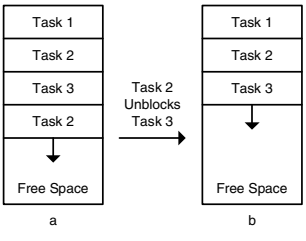

Fig. 14. Protothreads stack usage

4.5 Interrupts

Since interrupts can occur at any time, preemptive multithreads must accommodate for them by allocating extra space for the largest interrupt service routine (ISR). This extra overhead is also required for Y-Threads. A separate stack can be used for interrupt processing, which reduces the overhead required per thread to that required for context storage. The disadvantage of a separate interrupt stack is added processing overhead for the stack switch. A combination of separate and common stack processing for ISRs can be used to allow for good RAM usage while providing good performance for higher frequency ISRs. Having two types of ISRs reduces the ISR uniformity, making it a less ideal method.

Y-Threads already provide a method of processing on a common stack. This means that all ISRs can be written to execute on the current stack, and if the ISR needs extra space for processing it can invoke an RCR. The Y-Threading ISR programming paradigm is uniform, and provides the flexibility for either fast or large ISRs.

5 Future Work

The main areas of future work are related to development tools. Adding Y-Thread support to a language and the development of Y-Thread compatible libraries is a major area of future work. As stated in [18] threading must be part of the language for the best performance, and not just implemented in a library. This is true for Y-Threads as well. Another area of future work is to statically profile the Y-Thread software.

5.1 Language/Compiler Support

Language/Compiler support can improve the ease of programmability, and reduce the overhead of invoking Y-Thread RCRs. Currently Y-Thread RCRs are implemented using a helper function to call the RCR. The calling syntax is less than ideal since the function pointer and the data parameter must be passed to the *oss_rcr_call* helper function. Ideally the programmer would just invoke the RCR as if it were a normal function invocation, and the compiler would generate the stack switching wrapper necessary for the RCR call. If the language/compiler natively supported Y-Threads it would allow for RCRs with signatures other than a single void pointer with a void return type. Language/Compiler Support also would reduce the overhead of an RCR invocation. The data parameters which currently need to be copied twice would be copied directly to the RCR stack. The overhead of the *oss_rcr_call* function call would also be removed. Overall the added overhead of an RCR invocation could be reduced to checking if currently on the RCR stack, entering/exiting of an atomic state, and switching of the stack pointer. However, as stated in the Section 2, the compiler would need to have knowledge of the OS, which most likely would bind the language/compiler to a single RTOS.

5.2 Y-Threads and Libraries

The initial development of Y-Threads was driven by the observation that embedded systems often run a cycle of work and waiting. It was also observed that blocking is often done at the highest level. However not all blocking is done in this manner. If layers are built upon blocking calls then the dedicated Y-Thread stack may need to be larger than if blocking were to be done at a shallower location. The feasibility of using blocking libraries in Y-Threads must still be determined. Development of libraries using Y-Threads that block at higher levels is one possible option. It is possible that these libraries could be accessed using some form of message passing or shared memory, but work in this area is necessary to determine the practicality. The development of Y-Thread friendly libraries would be a logical extension of language/compiler support for Y-Threads.

5.3 Static Profiling

If a compiler existed that natively supported Y-Threads it could determine RCR stack requirements and if an RCR blocked. Determining the RCR stack requirements could improve memory utilization allowing for more RAM to be available for the heap, main stack and data space. If higher priority thread preemption is not allowed during an RCR invocation and there are no recursive function invocations or function pointer invocations within the RCR, then the memory requirements can be determined for the RCR stack. Either an error or a warning could be generated by the compiler if it detects that an RCR can invoke a blocking function since blocking is not allowed in an RCR. Statically determining if an RCR can invoke a blocking function or not only requires that no function pointer invocations exist within the RCR or its function invocations.

The RCR stack only needs to accommodate for the worst case stack utilization for all RCRs and any functions they invoke. If recursion exists within any of the RCRs then it will not be possible to determine the RCR stack requirements since the

recursion depth is bound at runtime[9]. Function pointer invocations are bound at runtime and therefore would make a compile time analysis impossible. If higher priority thread preemption is allowed then the number of simultaneous RCR invocations cannot be bound at compile time and therefore the RCR stack requirements cannot be determined.

Determining if an RCR can invoke a blocking function, is a simple matter of following all possible function invocations from the RCR. This is similar to compilers detecting dead code.

6 Conclusions

Y-Threads can be implemented easily on systems that currently support preemptive multithreading as is the case with OS*. The advantage of Y-Threads is that they provide a preemptive multithreaded programming model with good memory utilization.

Automatic allocation of memory on the RCR stack can be done with less overhead than dynamic allocation. The RCR call overhead is constant making it more desirable than dynamic memory allocation for systems with real-time requirements. The RCR call and automatic allocation overhead is slightly higher than statically allocated global buffer implementation without the static allocation of the memory. The RCR memory is available for use by any thread, where static global allocation of memory can only be used by a single thread. This need for having memory available for other threads is shown in the time synchronization application that had less than 41% of dedicated memory for its threads.

The hybrid of the two concurrency models in Y-Threads has allowed for the best of both models. Future work could further reduce overhead with language support. Language support could also add the ability to detect blocking in RCRs and the RCR stack requirements for further optimization and application correctness checking. The use of Y-Threads in embedded systems specifically WSN could reduce the time of application development when compared to event-based approaches. Y-Threads also have the potential to increase the capabilities of preemptive multithreaded systems due to better resource utilization.

Acknowledgments. This work is supported in part by NSF grants CNS-0435531, CNS-0520269, and EIA-0224469. The authors would also like to thank Joel Koshy and the anonymous referees for their insightful comments on an earlier draft of this paper.

References

[1] 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.

[2] J. Elson, Time Synchronization in Wireless Sensor Networks, PhD Dissertation, 2003.

[3] N. Reijers, K. Langendoen, Efficient code distribution in wireless sensor networks, Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications, 2003.

[4] S. S. Kulkarni, L. Wang, MNP: Multihop Network Reprogramming Service for Sensor Networks, 25th IEEE International Conference on Distributed Computing Systems, 2005.
[5] J. Hui, D. Culler, The Dynamic Behavior of a Data Dissemination Protocol for Network Programming at Scale, Proceedings of the 2nd international conference on Embedded networked sensor systems, 2004.
[6] J. Hill, R. Szewczyk, A. Woo, System Architecture Directions for Network Sensors, Architectural Support for Programming Languages and Operating Systems, pages 93–104, 2000.
[7] C. Han, R. Kumar, R. Shea, A Dynamic Operating System for Sensor Nodes, Proceedings of the 3rd international conference on Mobile systems, applications, and services, 2005.
[8] H. Lauer, R. Needham, On the Duality of Operating System Structures, Proceedings of the Second International Symposium on Operating Systems, IRIA, 1978.
[9] R. Behren, J. Condit, E. Brewer, Why Events Are a Bad Idea (for high concurrency servers), 9th Workshop on Hot Topics in Operating Systems (HotOS IX), 2003.
[10] J. Ousterhout, Why Threads Are a Bad Idea (for most purposes), Invited talk given at USENIX Technical Conference, 1996.
[11] T. Baker, Stack-Based Scheduling of Realtime Processes, Journal of Real-Time Systems, 3, 1991.
[12] OSEK/VDX Operating System Version 2.2.3, available at http://osek-vdx.org/mirror/os223.pdf, 2005.
[13] J. Koshy, R. Pandey, Remote Incremental Linking for Energy-Efficient Reprogramming of Sensor Networks, Proceedings of the Second European Workshop on Wireless Sensor Networks, 2005.
[14] Dunkels, O. Schmidt, and T. Voigt, Using Protothreads for Sensor Node Programming, Proceedings of the REALWSN'05 Workshop on Real-World Wireless Sensor Networks, 2005.
[15] Dunkels, B. Gronval, T. Voigt, Contiki – a Lightweight and Flexible Operating System for Tiny Networked Sensors, Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks, 2004.
[16] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, J. R. Douceur, Cooperative Task Management without Manual Stack Management or, Event-driven Programming is Not the Opposite of Threaded Programming, Proceedings of the 2002 USENIX Annual Technical Conference, 2002.
[17] S. Goldstein, K. Schauser, E. Culler, Lazy Threads: Implementing a Fast Parallel Call, Journal of Parallel and Distributed Computing, 1996.
[18] H. Boehm, Threads Cannot Be Implemented as a Library, Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation, 2005.