Stackless Preemptive Multi-Threading for TinyOS

William P. McCartney and Nigamanth Sridhar
Electrical and Computer Engineering
Cleveland State University
Cleveland OH 44115 USA
Email: {w.p.mccartney,n.sridhar1}@csuohio.edu

***Abstract*—Programming support for multi-threaded applications on embedded microcontroller platforms has attracted a considerable amount of research attention in the recent years. This paper is focused on this problem, and presents UnStacked C, a source-to-source transformation that can translate multithreaded programs into stackless continuations. The transformation can support legacy code by not requiring any changes to application code, and only modifications to the underlying threading library. We describe the details of UnStacked C in the context of the TinyOS operating system for wireless sensor network applications. We present a modified implementation of the TOSThreads library for TinyOS, and show how existing applications programmed using TOSThreads can be automatically transformed to use stackless threads with only modifications in the build process. By eliminating the need to allocate individual thread stacks and by supporting lazy thread preemption, UnStacked C enables a considerable saving of memory used and power consumed, respectively.**

I. Introduction

In this paper, we focus on multithreading approaches for programming wireless sensor network systems. In particular, we present techniques aimed at TinyOS [7], a widely popular operating environment specifically designed for building wireless sensor networks. TinyOS is a component-oriented environment that presents a core set of components that applications can "wire in", and then add application behavior that operates on top of these core components. Applications for TinyOS are written using the nesC programming language [5], which is collection of syntactic constructs that provide a component-based programming view to C programming. Traditionally, nesC programs have been written using the event-driven programming paradigm. As such, the main constructs in nesC that enable concurrency are *tasks* and *events*. Application behavior is spread across tasks and events with the goal of reducing the amount of time spent in blocking input/output operations (*e.g.,* radio messaging, sensing).

While such an event-driven view is very appropriate during application execution, the view is not appropriate as a *programming methodology* — the cognitive load involved in *writing* event-driven programs is high. Over the last few years, there have been many proposals for introducing constructs for TinyOS that will enable programmers to develop applications using multi-threading. In previous work, we designed *TinyThread* [9], a library solution to enable *cooperative* multi-threading for nesC. The two primary limitations of this solution were: (a) it was a library solution that existed outside of the default toolchain, and (b) cooperative threading requires a programmer to be careful in not designing threads that hog the processor for too long. As an improvement to this work, Klues et al [8] presented *TOSThreads*, a multi-threading system that was fully integrated into the compile toolchain, as well one that allowed for thread preemption.

Event-driven programs are still very attractive from a resource utilization perspective. Because of the fact that the programmer has to manually deal with maintaining state across tasks, each individual task does not need maintain its own stack. By contrast, in multi-threaded programs, the stack is managed automatically. The downside of this is that the memory requirements of threaded programs are usually much larger than their event-driven counterparts, since each thread has to allocate its maximum possible required stack during its entire existence. Systems that have low memory resources –such as those typically used for WSNs, with only a few kilobytes of RAM– present a dilemma. It would be nice to combine the programming expressiveness of multi-threading with the resource effectiveness of event-driven execution.

In this paper, our primary contribution is a way of resolving this dilemma. We present an implementation of TOSThreads that transforms multi-threaded code into *stackless* threads. The transformation is done automatically using a source-to-source translator (called UnStacked C). The translator converts regular C code that uses threads into *tasklets*. Since these tasklets follow event semantics, they do not need a separate stack, and hence their memory overhead is substantially reduced. At the same time, the readability of the program is not compromised: programmers can use regular C constructs. Our system enables programmers to write applications in a multi-threaded fashion and then have them execute in an event-driven manner.

Consider, as a case study, the comparison of programming systems for wireless sensor networks. The nesC programming language is fully event-based. The programmer is faced with building applications as event-driven state machines. As such, the programmer is immediately burdened with "stack ripping" problems, and managing state transfers among tasks and events is hard. At the same time, however, the TinyOS execution model is extremely efficient, and is very attractive given the resource-constrained hardware platforms that are targets for these programs. (The MicaZ platform has 4K of RAM, and the TelosB platform has 10K). In our comparison, we can place nesC at the bottom-right: very memory-efficient, but poor in program expressiveness.

At the diametrically opposite corner lie threading ap-

Execution Model	Blocking I/O	Local Variables	Multiple Context	Preemption	Stackless	Data Integrity
Event Driven		✓			✓	✓
ProtoThreads	✓		✓		✓	✓
Cooperative Multithreading (TinyThreads)	✓	✓	✓			✓
Preemptive Multithreading (TOSThreads	✓	✓	✓	✓		
UnStacked C	✓	✓	✓	✓	✓	✓

Fig. 1. Features of Multi-threading Models

proaches with fixed-size stacks such as TOSThreads [8], TinyThreads [9], and MANTIS [2]. These approaches enable applications to be developed using multi-threading, where each thread has its own stack to store its context. This approach, while greatly improving the expressivity of programs, is not too attractive when considering memory efficiency: the memory footprint is considerably larger than that of the equivalent nesC program. Protothreads [4] provide a better alternative in terms of memory efficiency — each thread only requires 2 bytes of memory. However, protothreads are not as expressive as regular threads.

In comparison to these models, UnStacked C presents a nice "sweet spot". In terms of memory efficiency, UnStacked C programs are quite close to nesC and protothreads, and in terms of program expressiveness, UnStacked C is identical to a true preemptive multi-threading model. We think that UnStacked C can reveal new design strategies for programs running on small resource-constrained embedded systems.

Figure 1 shows several different threading models compared with UnStacked C. Not only does UnStacked C support a reduced RAM footprint, but it does this without compromising features that users are accustomed to. This includes data integrity. Any time normal preemption is used, data faults between preempted threads can occur. Preemption is desirable since it can allow users to have long running computations without damaging response time. In UnStacked C we use *lazy preemption* that forces any pending writes to occur prior to switching to a new context. Through this technique, UnStacked C achieves the data concurrency of event-driven systems with the ability to preempt the current execution. Further details of preemption can be found in Section III-A.

With UnStacked C legacy preemptive and cooperative multithreaded C programs can be recompiled with some minor modifications to the underlying framework. Modifications are typically required of the underlying threading framework, not of the individual applications. For example, our implementation of wrappers for the TOSThreads API [8] allows us to recompile TinyOS applications written using TOSThreads with almost no changes to application code.

The rest of the paper is organized as follows. In Section II, we present an overview of related work, and set up the context for our work. Section III describes details of the TOSThreads implementation using UnStacked C. We present some analysis of our UnStacked C implementation in Section IV, and present results from performance evaluation when comparing UnStacked C to TOSThreads in the TinyOS context in Section V. We conclude in Section VI.

II. Related Work

Several systems have been proposed that challenge the TinyOS position on events vs. threads for building embedded system applications. *Protothreads* [4] are a way of programming embedded systems running the Contiki operating system [3] using a limited form of stackless threads. There are a few large limitations of protothreads. First, protothreads require programmers to use a special, non-standard syntax. Second, protothreads cannot have any arguments or local (automatic) variables, only global variables. Third, protothreads cannot simply call a blocking routine, they can only spawn off another protothread and wait for said protothread to complete. Since there is no scheduling framework, the developers must call protothreads when an event handler occurs. These restrictions require great discipline from the developer. In spite of this disadvantage, the big advantage of protothreads over other threading solutions for embedded systems is extremely small memory footprint. UnStacked C uses a similar implementation technique as that used by protothreads.

There are other attempts at compiler centric stackless threading for sensornets, namely *Threads2Events* [1] and *TinyVT* [13]. Threads2Events performs a transform that is similar to UnStacked C in that it creates a context to store the local variables and child calls. Instead of passing a reference to the context it uses global variables to store them limiting a single instance of any one thread. If a blocking method is called from multiple threads, it must be indirectly mapped to the calling function. It requires users to make additions to the Platform Abstraction Layer inside of the compiler to support any blocking primitives. Similar to Tame, it cannot recompile existing applications.

TinyVT [13] is another TinyOS centric thread to event compiler which adds cooperative threading. TinyVT is designed to allow users to write their TinyOS event driven code, in a procedural fashion. It enables users to write a single function with multiple wait statements and have it be transformed into multiple events. This does not enable building of functional primitives nor hiding the event driven system.

TinyThread [9] is a full-functional threading API for TinyOS that enables programmers to write cooperatively-threaded programs. This library is much more heavyweight than protothreads since each thread requires its own stack. Stacks in TinyThread are automatically managed, which means that programs can use local variables, and high-level synchronization constructs across threads. Although the thread library is accompanied by a tool that provides tight estimates of actual stack usage, the memory requirement places a severe limitation on the number of threads that can be accommodated on typical sensor hardware.

```
1  async command error_t ThreadScheduler.suspendCurrentThread(){
2    atomic {
3      if(current_thread->state == TOSTHREAD_STATE_ACTIVE) {
4        current_thread->state = TOSTHREAD_STATE_SUSPENDED;
5        suspend(current_thread);
6        return SUCCESS;
7      }
8      return FAIL;
9    }
10 }
```

Fig. 2. Original TOSThread `suspendCurrentThread` method

```
1  void yield() attribute((blocking,yield)){
2  }
3
4  async command error_t ThreadScheduler.suspendCurrentThread(){
5    atomic {
6      if(current_thread){
7        if(current_thread->state == TOSTHREAD_STATE_ACTIVE){
8          current_thread->state = TOSTHREAD_STATE_SUSPENDED;
9          yield();
10         return SUCCESS;
11       }
12     }
13     return FAIL;
14   }
15 }
```

Fig. 3. UnStacked C `suspendCurrentThread` method

TOSThreads [8] is the de facto standard threading model for TinyOS. It is a preemptive threading library for TinyOS. It is included with TinyOS 2.1 and supports a wide variety of hardware. It includes static and dynamic threading along with both a nesC and C interfaces. It currently does not include any stack sizing so the memory overhead can be significant and uncertain. We have an updated TOSThreads UnStacked C implementation that *significantly* reduces the memory overhead while retaining its flexibility.

Y-Threads [12] is a lightweight threading system which attempts to break each thread into two separate stacks. The first stack is the blocking portion of the thread, and the second part is the non-blocking or shared version of the stack. Since the shared portion of the program does not block, no stack storage is required. This inspired the *blocking* attribute in UnStacked C. In contrast with UnStacked C, Programmers must select which portions of the program are blocking (and therefore require stackspace).

Shared-stack cooperative threads [6] are particularly close in spirit to UnStacked C. Shared-stack threads operate exactly like regular cooperative threads, except that all threads execute on the same system stack. When a thread blocks, it pushes the registers onto the stack, and then copies that stack to elsewhere. To resume a thread, its stack gets copied back into the system stack, then the registers get popped and execution continues. UnStacked C operates in a similar fashion in terms on only storing the blocking portion of the continuation, but without the stack copying and explicit register operations.

Yang, et. al. have a source-to-source translation that reduces the overall memory of embedded systems by an aggressive non-typical inlining technique [15]. Instead of force inlining all functions, which would can create multiple copies of the same function, they copy the routines into one larger routine and use goto statements to enter and exit the equivalent portions of the function. They perform this translation as an attempt to "flatten" all of the functions into main, and "lift" local variables into global memory space. Their transform can reduce or eliminate the stack of a single program without threads, but it does not eliminate the need for multithreaded programs to have a separate stack for each thread. This work is orthogonal to our work, and can be applied before or after our transformation as an optimizer.

III. TOSThreads Implementation

UnStacked C uses a C-to-C translator to convert translate multi-threaded code into stackless threads. Our implementation of this translation is built using the CIL translation framework [11]. The entire transform could have been implemented in CIL directly, but we instead relied on patches to serialize the data prior to our processing [10]. The translator loads the entire application into memory before the abstract syntax tree is manipulated. The transformation from multi-threaded code into stackless threads is achieved by way of a series of abstract syntax tree modifications.

A key feature of UnStacked C is that existing application code need not be modified at all in order to use stackless threads. Therefore, applications programmed as multi-threaded applications using a threads library are automatically able to use the advantages of UnStacked C. The underlying thread library, on the other hand, does need to be modified. The primary pieces of the thread library implementation that need to be modified have to do with how the thread library manages thread stacks. The existing processor specific stack-swapping routine will need to be replaced with a platform agnostic routine. Another major modification involves how contexts are allocated (or at least the sizes of contexts). In the remainder of this section, we describe the details of our modifications of the TOSThreads library to enable TinyOS code written using this library to be transformed into stackless threads.

These modifications are all implemented through the use of function attributes. The first attribute is used to identify a *yield* function. A function that is identified as a yield function does not need to be named `yield`, but it must be annotated with the `yield` attribute. Any calls to the yield function will force the thread to return to the scheduler. Calls to this routine are actually replaced with a guaranteed yield. The yield function is expected to take no arguments and have no return value. Notice that, in contrast to cooperative threading systems, this yield function is not actually called in application code but is only called from within the thread library.

Figure 2 shows the command `suspendCurrentThread()` in the default implementation of TOSThreads. This routine is called in all of the blocking routines and during preemption to force the current thread off of the stack and to allow the scheduler to start executing the next thread. In order to modify this for UnStacked C we added a yield routine and replaced the `suspend()` call (Figure 2, line 5) with a call to `yield()` as shown in Figure 3 (line 9). The original

```
void interruptThread() __attribute__((noinline)) {
  if(call ThreadScheduler.wakeupThread(
      TOSTHREAD_TOS_THREAD_ID) == SUCCESS)
    if(call ThreadScheduler.currentThreadId() !=
        TOSTHREAD_TOS_THREAD_ID)
      call ThreadScheduler.interruptCurrentThread();
}

inline async command void PlatformInterrupt.postAmble() {
  atomic {
    if(call TaskScheduler.hasTasks() == TRUE )
      interruptThread();
  }
}
```

Fig. 4. Original TOSThread interrupt postamble.

```
async command error_t
    ThreadScheduler.interruptCurrentThread() {
  atomic {
    if(current_thread->state == TOSTHREAD_STATE_ACTIVE) {
      current_thread->state = TOSTHREAD_STATE_READY;
      call ThreadQueue.enqueue(&ready_queue, current_thread);
      interrupt(current_thread);
      return SUCCESS;
    }
    return FAIL;
  }
}
```

Fig. 5. Implementation of `interruptCurrentThread` in the default implementation of TOSThreads

`suspend()` function schedules the next thread for execution and swaps stacks, or puts the processor to sleep. Instead in UnStacked C, we add this functionality to the task scheduler to run threads when the queue is empty, or to put the processor to sleep without swapping stacks.

TOSThreads already has a scheduler implemented (currently round robin) and UnStacked C uses this same scheduler to choose which thread to execute (if any). Two modifications had to be made to this scheduler in order to support stackless threads. The first change is to make the system TinyOS thread to have a different status than regular threads, and therefore stay outside of the purview of the scheduler. This actually simplifies the main routines, since the scheduler does not have to be initialized until the first actual thread needs to be scheduled. The second major modification was to simplify how preemption and interrupts are handled.

Normally in multi-threading systems stack-swapping routines must be written in assembly for each platform. Since UnStacked C never switches stacks, there is no platform-specific assembly code. In that respect, once a multi-threading system has been ported to use UnStacked C, it can run on any hardware platform. For TinyOS this is particularly important since there are so many platforms that are supported.

A. Preemption

Preemption and interrupts are typically a complicated part of any threading system. In TOSThreads this is doubly true since the threads have a lower priority to the tasks. This means that if a task is posted from an interrupt, the stacks must be swapped to execute the main thread. TOSThread adds a postamble to each of the interrupts which performs this task. The original postamble is shown in Figure 4. When an interrupt fires, if a task has been posted, it tries to wake up the TinyOS thread (if it

```
void interruptThread() __attribute__((noinline)) {
  call ThreadScheduler.interruptCurrentThread();
}

inline async command void PlatformInterrupt.postAmble() {
  atomic {
    if(call TaskScheduler.hasTasks() == TRUE )
      interruptThread();
  }
}
```

Fig. 6. TOSThread interrupt postamble modified for UnStacked C.

```
async command error_t
    ThreadScheduler.interruptCurrentThread() {
  atomic {
    if(call TaskScheduler.hasTasks() == TRUE){
      preempt_flag = 1;
    }
    return SUCCESS;
  }
}
```

Fig. 7. Implementation of `interruptCurrentThread` in the UnStacked C TOSThreads library

is not already active) and to switch to that thread. The original TOSThreads implementation of `interruptCurrentThread` can be found in Figure 5 and it forces a stack swap and context change to occur immediately.

In contrast to original TOSThreads, when modified for UnStacked C the interrupt postambles become simpler. Preemption in the UnStacked C implementation of TOSThreads is done in a lazy fashion. Instead of forcing a stack swap and context change every time an interrupt is done executing, when a thread is interrupted, a flag is simply set to notify the scheduler that the thread can be preempted when needed. To support this, the compiler must generate a test point in many places throughout any blocking routines. This test point checks to see if the preemption flag has been set, if so then it must return. These test points are added at the ends of loops and `goto`s which branch backwards. They are only added to blocking functions but they do add to the overall program size.

Lazy preemption has other benefits besides not requiring separate stacks. Lazy preemption helps maintain data integrity. For instance, in a program with two threads simultaneously reading and writing a variable with preemption, data faults can occur. Since preemption cannot occur in the middle of reading or writing a variable, the same data faults cannot occur in UnStacked C. Normally in preemptive threads, preemption can occur at anytime, including in the middle of a single multibyte addition. This means that the thread does not have a finite number of states, or at least not from the programmers perspective from the source code. In UnStacked C preemption can occur only at a test point, so the number of different possible states a given thread could be in is a finite number. This means that UnStacked C reduces each thread to a finite state machine, and also reduces the overall number of states in the entire system. This can make the system easier to undestand.

This is shown in Figure 6. Since we are using lazy preemption, there is no harm in calling `interruptCurrentThread()` excessively, so several of these checks do

```
event void PreemptionAlarm.fired() {
  call PreemptionAlarm.startOneShot(
      TOSTHREAD_PREEMPTION_PERIOD);
  atomic {
    if((call ThreadQueue.isEmpty(&ready_queue) == FALSE)) {
      call ThreadScheduler.interruptCurrentThread();
    }
  }
}
```

Fig. 8. Original implementation of the TOSThread `PreemptionAlarm.fired` event

```
event void PreemptionAlarm.fired() {
  call PreemptionAlarm.startOneShot(
      TOSTHREAD_PREEMPTION_PERIOD);
  atomic preempt_flag = 1;
}
```

Fig. 9. Updated UnStacked C implementation of the `PreemptionAlarm.fired` event

not need to be in place. The UnStacked C code for `interruptCurrentThread()` can be found in Figure 7: it simply sets a preemption flag to be checked in the threads.

Preemption among threads in TOSThreads is handled similarly. The software timers in TinyOS are utilized to implement the `PreemptionAlarm` which forces a swap between threads. The original code is shown in Figure 8. This function checks to see if there is another thread ready to execute, and then task swaps to the next thread to execute. Since these software timers are not run inside of interrupts (but from tasks) they are only there to handle the case when no interrupt postamble exists. Comparably, we implemented the UnStacked C version in a similar fashion to the interrupt postamble containing only enough to set the preemption flag. This version of `PreemptionAlarm` for UnStacked C is shown in Figure 9.

When dealing with thread preemption, it is important to ensure that faults are not accidentally introduced in the code. Our UnStacked C compiler can detect and warn the user of certain faults in their application code. One common fault in these mixed (multi-threaded and event-driven) applications occurs when a developer mistakenly calls a blocking routine from an event or a task. This normally causes a system fault, but with UnStacked C, our compiler will error out, notifying the user of the problem at compile-time. In UnStacked C, the opposite problem, calling event-driven code inside of a thread, is perfectly natural so it does not create a fault, but in regular TOSThreads this could cause a fault due to preemption.

B. Blocking

The next modification in the UnStacked C implementation of TOSThreads is to mark blocking functions with the `blocking` attribute. This attribute marks a function as needing to be transformed into a stackless routine. This attribute is not normally needed, since any routine that calls `yield` gets marked as `blocking`. Also any routine that calls any other routine that is marked as `blocking` is also marked as `blocking`. To accomplish this the call graph is processed repeatedly until no new blocking calls are found. This means that any routines that perform any long running processing should be marked as blocking so that they can be preempted.

```
uint8_t stack[stack_size];
```

Fig. 10. Original TOSThread stack allocation code where stack_size is a parameter that the developer estimates as the stack size.

```
//The code the programmer writes
#define STACKSIZE(NAME)  int __attribute__((blockingstack(NAME)))
STACKSIZE(run_thread) threadstack;

//The code after translation
struct UnStacked_run_thread threadstack;
```

Fig. 11. Updated UnStacked C context allocation code, where the compiler calculates the exact required RAM for a function named run_thread.

Normally developers must make an educated guess how much stack space a given thread will require, and then allocate that much space for each instance as shown in Figure 10. Instead of this, in UnStacked C a context is allocated for each instance. This context is calculated by the compiler after it computes a given routine's call graph. It then generates a structure for the particular function. Ideally the developer would simply allocate the size of the structure for each instance of a thread. Since the structure does not exist in the program the developers write (it is added by the compiler after the fact) the developers need a way to have the compiler allocate the correct amount of memory. To assist developers allocating exactly the RAM they need, we added the `blockingstack` attribute.

When a variable is marked with the `blockingstack` attribute, the compiler looks at the arguments of blockingstack attribute and then changes the type of that variable into the structure of the type. In a static configuration, this allows variables to be transformed into the contexts. In a dynamic configuration, this can be used by a macro to calculate the required context size for each function, or to calculate the maximum context size for a number of functions simultaneously at compile time. The transform the compiler performs can be seen in Figure 11.

The other modifications to the original source code are to remove any actual stack swapping and stack allocation routines. These remove more source code than adding new code. In our UnStacked C TOSThread implementation, we actually ended up with 450 fewer lines of code than the original implementation.

C. Limitations

The current version of UnStacked C has four main limitations. All of these limitations only impact the functions which are blocking. If the rest of the program contains any of these they will not be impacted, only the blocking multi-threaded code.

The first limitation is that the entire program must be compiled together at once since the transform is achieved through whole-program compilation. A good example of a problem occurring is with `printf()`. `printf` in many embedded implementations requires a callback to operate and if that callback is a blocking operation, then `printf` needs to be a blocking function also. Since `printf` is typically precompiled, that means it needs to be brought into the project and then recompiled by UnStacked C.

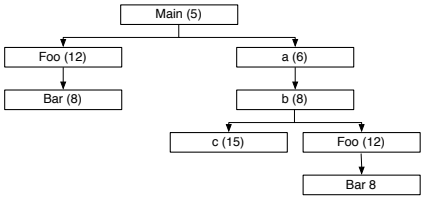

Fig. 12. A simple call graph. Each node is labeled with the name of the called function, with its own stack overhead in parentheses.

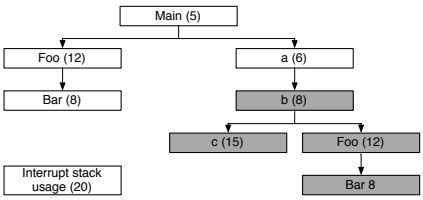

Fig. 13. A simple call graph to illustrate computation of maximum stack depth in the presence of interrupts.

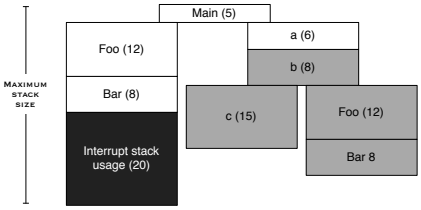

Fig. 14. Calculation of stack depth in a call graph with interrupts enabled

The second limitation is that handling of indirect calling of blocking routines must be done with care. If someone invokes a blocking routine it is assumed that they are using the correct signature. This means that the caller is providing a child context for it to execute it. In most embedded systems this is only done by the scheduler (which already needed to be modified to support UnStacked C) so it is not a problem.

The third limitation is that recursion is not supported. It is possible to implement recursion in a system as long as the maximum depth of the recursion is known. It is very common for embedded system developers and embedded software standards to disallow recursion in embedded software so this is also not a critical issue.

The last limitation is that UnStacked C does not support dynamic linking and loading of applications. Since the entire source code is compiled together, run-time components cannot be added.

IV. Analysis of Stack Allocation

To explain why our transformation to UnStacked C saves RAM, we must first explain the RAM allocation of different programs. We will compare event driven, threaded, and UnStacked C programs. Figure 12 shows a generic call graph. Each node represents a function call. The total stack depth is not determined solely by the number of calls, but by the sum of the function call overheads summed to the maximum possible amount.

Figure 12 has each function labeled with its own overhead. To find the stack requirements of a function, simply add worst case stack requirements to the given functions stack overhead. In this case, the deepest possible stack depth is calculated by $Main(5) + a(6) + b(8) + Foo(12) + bar(8) = 39$.

For any given thread, including the main thread in event-driven programs, to calculate the maximum stack required the call graph alone is not enough, since interrupts can occur. Figure 13 adds a simplified interrupt stack overhead (which is a call graph in and of itself) and the white nodes are when interrupts are enabled. The new calculation is shown in Figure 14 which gives a greater maximum stack depth.

Each thread needs a continuous block of RAM allocated for the worst case stack requirements. The average stack usage is irrelevant since any of the running threads may hit their maximum stack depth at some time during execution. Since a call instruction simply pushes the program counter onto the stack and then begins executing the next function, each function's stack overhead must be allocated in contiguously.

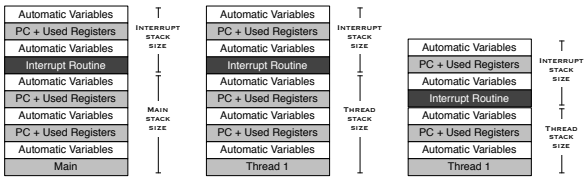

Fig. 15. Stack depth computation in a multi-threaded application

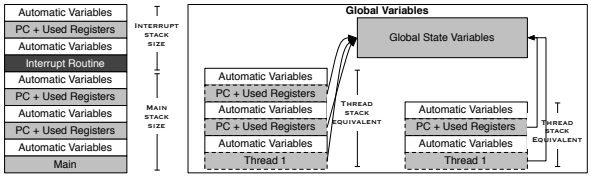

Fig. 16. Stack depth computation and memory from "stack ripping" in an event-driven application

A function's stack overhead is made up of the automatic variables and the program counter. The program counter is the location that will be resumed when the function returns. Any other used registers must be stored on the stack too.

Since a given thread's stack size is determined by the worst case stack consumption. That worst case stack can be broken down into the automatic variables and the registers as shown in Figure 15. It is important to notice how each thread requires an overhead for the interrupts.

Figure 16 shows an equivalent algorithm written in an event driven format. In an event-driven system the PC and registers are not used to store the state of each of these state machines, instead global state variables are used. The automatic variables that need to be stored across events must also be stored as globals. In many event-driven systems, programmers have optimized the system by storing the equivalents of automatic variables globally only if they are used in multiple events.

In our transforms, instead of storing the PC we store a state index, but similarly to threaded stack we still store the automatic variables (Figure 17). Notice that in both the stackless (Figure 17) and the event driven (Figure 16) RAM usage by each of the threads can be reduced significantly.

V. Performance Evaluation

TOSThreads [8] is a threads library that allows developers to write multi-threaded applications for the TinyOS platform, with full support for preemption. When using this library, developers enjoy the convenience of programming procedural

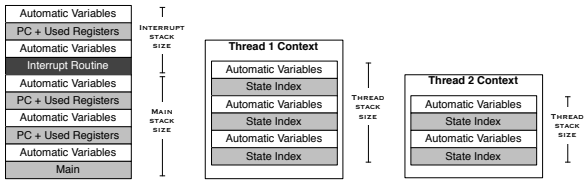

Fig. 17. Stack depth computation in a multi-threaded application using UnStacked C

code, and yet enjoy the performance benefits of event-based execution at the lower levels of the operating system.

For our evaluation, we used seven applications that are in the default TOSThreads distribution [14]; we did not implement any new applications for this comparison. The use of UnStacked C does not require any change at the application level. As such, for our evaluation, we did not modify the application code at all for any of the test applications. Instead, we only modified the implementation of the threads library. Instead of using the TOSThreads library, we modified the library, and apply the UnStacked C program transformation. The only change in the workflow is a different invocation to the build system. The tests were all conducted on the same compiling host with TinyOS 2.1.0 and GCC 3.2.3.

1) Memory Usage: Figure 18 shows the RAM usage of all the applications we evaluated. The RAM usage in applications using regular TOSThreads is broken up into the memory used by the base application and that used by for the thread stack(s). In every case, the UnStacked C implementation of TOSThreads yields a smaller RAM footprint, on average a reduction of 35%. This reduction is primarily because of the fact that in the UnStacked C implementation, there are no dedicated stacks allocated for each thread. The reduction in memory utilization is directly proportional to the number of threads that an application uses.

When programming using TOSThreads, a developer is forced to guess the stack size of each thread. If the wrong size is chosen the stack could overflow into other variables or another stack (creating a system fault). If too much memory is selected, then that memory is wasted. In these applications the stack sizes of the threads (100-800 bytes) and the numbers of threads (1-6) varied. Since the stack sizes vary so much for each application, we assume that much care was given (through simulation) to figure out the optimum stack size for each of these examples.

Since UnStacked C can only reduce the stack requirements of the threads, Figure 18 also shows both the RAM usage of the original application and the stack consumption. That also includes the RAM for the UnStacked C stackless compilation. On average, the stack usage of the original TOSThreads can be reduced by more than 80% on average.

This reduction in RAM usage comes at a cost, since additional code is generated. The ROM (flash) comparison is shown in Figure 19. The overhead of the ROM is only a 12% increase on average. The ROM increase is expected since multiple entry and exit points were added to each blocking function. Among those entry and exit points are the test points

Multi-threading Approach	ROM	RAM	CPU usage
TOSThreads	5234	1002	11.58%
TOSThreads (UnStacked C)	6176	769	0.68%
ProtoThreads	3189	66	0.54%
Event-Driven	2590	55	0.42%
TinyThread	3626	296	0.67%
TinyThread (UnStacked C)	3760	152	0.72%

TABLE I
COMPARING RESOURCE UTILIZATION FOR THE *Blink* APPLICATION USING DIFFERENT MULTI-THREADING APPROACHES

added due to the lazy preemption. TOSThreads normally relies on pushing all of the registers onto the stacks instead of these entry/exit points. This contrasts with UnStacked C that creates code to the store the values in an orderly fashion.

2) Power Consumption: In most sensing applications, the CPU on the sensor node sleeps for the majority of the time. It is important that a threading system does not create overhead which will can draw too much power from the rest of the system. This overhead is generally caused by interrupts, and the threading system's ability to rapidly swap tasks.

We measured power consumption on a TelosB mote using a modified version of the TOSThread Blink application from the TinyOS distribution. The original Blink application toggles the LEDs on the mote based on a set of timers. In order to measure the current consumption of the CPU accurately with no influence from the LEDS, we modified the application to keep the LEDs off, and only have the threads get scheduled, but do nothing. This forces the same assembly code generation (it still writes the I/O port in a timely fashion), but it never turns the LED on. Since the LEDs are off, we can directly measure the MCU's power consumption. We compared the results for original TOSThread and UnStacked C and the results are shown in Figure 20. This shows how the microcontroller consumes 60% less when UnStacked C is used. From the graph, it is apparent that the interrupt overheads added to the system are impacting the power consumption. Since UnStacked C needs little interrupt overhead due to its lazy preemption, it saves a significant amount of energy. This is particularly important for long-running sensornet applications.

Figure 21 shows an up-close view of the current consumed by a single thread execution. This involves a timer waking up the scheduler. The scheduler then wakes up a threads and switches context to the thread. After the thread does any processing it needs to, it switches contexts back and goes to sleep. Ignoring the secondary impulses, the area under the curve is actually slightly smaller in the UnStacked C meaning that even without the interrupt overheads, the speed of context switching can save even more power.

Table I presents the resource utilization when using different multi-threading strategies for the Blink application. As can be seen from this table, the RAM usage is reduced by about 30% when using the UnStacked C implementation of the thread libraries (both TOSThreads as well as TinyThread). The CPU Usage is even more drastic: the power consumed by the UnStacked C implementation is an order of magnitude lower, attributed largely to the lazy preemption scheme that UnStacked C uses.

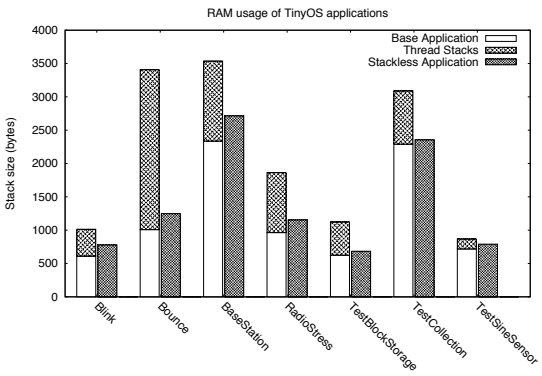

Fig. 18. RAM usage: TOSThreads vs. UnStacked C

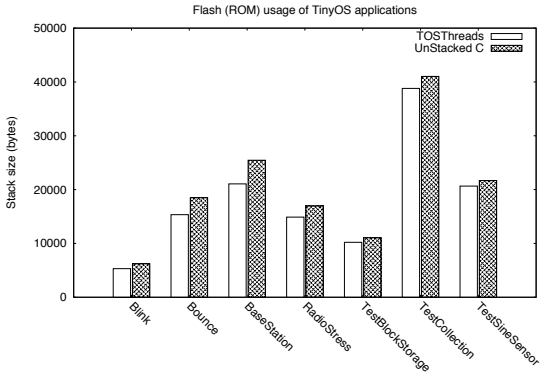

Fig. 19. Flash usage: TOSThreads vs. UnStacked C

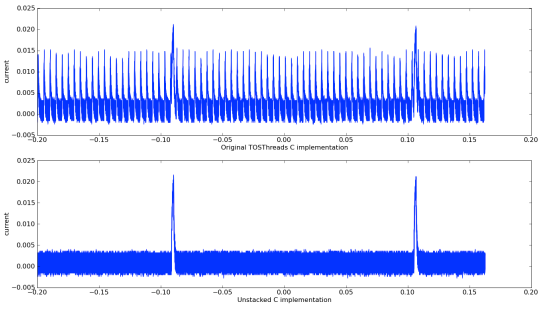

Fig. 20. CPU Current Consumption of Blink in TOSThreads vs. UnStacked C

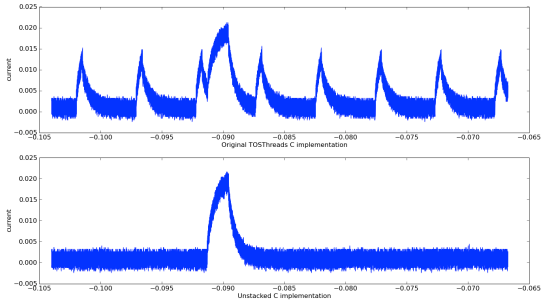

Fig. 21. CPU Current During Thread Execution Consumption in TOSThreads vs. UnStacked C

VI. Conclusions

In this paper, we have presented UnStacked C, a C-to-C translation approach to building stackless C continuations. The most important contribution of the work presented here is that it enables richer design strategies that were previously too "cost-prohibitive" in terms of memory utilization. Current users of TOSThreads and similar libraries can immediately benefit with lower memory footprint and power consumption in existing applications. The UnStacked C translator that we have implemented takes as input a C program written using preemptive threads and automatically generates the corresponding program that is an event-driven state machine.

We see UnStacked C as an enabler to designing richer functionality on low-resource devices. Sensor nodes, for example, that have been designed to be dumb data collectors because of the expressiveness limitations of the programming model can now be armed with extra functionality. We have demonstrated the viability of this by providing an alternate implementation of the TOSThreads thread library for TinyOS. We have shown a considerable reduction in memory usage as well as energy usage when applications are written using UnStacked C as opposed to using regular TOSThreads. All this is accomplished with no code modifications at the application code level. The applications are simply recompiled with the new implementation of the threads library in order to enjoy the cost savings provided by UnStacked C.

References

[1] A. Bernauer, K. Römer, S. Santini, and J. Ma. Threads2events: An automatic code generation approach. In *Proc. HotEMNETS 2010*, Killarney, Ireland, June 2010.

[2] S. Bhatti et al. Mantis os: an embedded multithreaded OS for wireless micro sensor platforms. *Mob. Netw. Appl.*, 10(4):563–579, 2005.

[3] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proc. LCN '04*, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.

[4] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *Proc. SenSys '06*, pages 29–42, New York, NY, USA, 2006. ACM.

[5] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Proc. PLDI '03*, pages 1–11, New York, NY, USA, 2003. ACM.

[6] B. Gu, Y. Kim, J. Heo, and Y. Cho. Shared-stack cooperative threads. In *Proc. SAC '07*, pages 1181–1186, New York, NY, USA, 2007. ACM.

[7] J. Hill et al. System architecture directions for networked sensors. In *Proc. ASPLOS-IX*, pages 93–104, New York, NY, USA, 2000. ACM.

[8] K. Klues et al. Tosthreads: thread-safe and non-invasive preemption in tinyos. In *Proc. SenSys '09*, pages 127–140, New York, NY, USA, 2009.

[9] W. P. McCartney and N. Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In *Proc. SenSys '06*, pages 167–180, New York, NY, USA, 2006. ACM.

[10] J. A. Meister et al. Serializing c intermediate representations for efficient and portable parsing. *Softw. Pract. Exper.*, 40(3):225–238, 2010.

[11] G. C. Necula et al. Cil: Intermediate language and tools for analysis and transformation of c programs. In *Proc. CC '02*, pages 213–228, London, UK, 2002. Springer-Verlag.

[12] C. Nitta et al. Y-Threads: Supporting concurrency in wireless sensor networks. In *Proc. DCOSS '06*, pages 169–184, jun 2006.

[13] J. Sallai, M. Maróti, and A. Lédeczi. A concurrency abstraction for reliable sensor network applications. In *Proc. 12th Monterey conference on RSUNP*, pages 143–160, Berlin, Heidelberg, 2007. Springer-Verlag.

[14] TinyOS Community. Tinyos website. http://www.tinyos.net.

[15] X. Yang, N. Cooprider, and J. Regehr. Eliminating the call stack to save ram. In *Proc. LCTES '09*, pages 60–69, New York, NY, USA, 2009.