10-2015

Stackless Multi-Threading for Embedded Systems

William P. McCartney  
*Cleveland State University*, billmccartney@gmail.com

Nigamanth Sridhar  
*Cleveland State University*, n.sridhar1@csuohio.edu

Follow this and additional works at: [https://engagedscholarship.csuohio.edu/enece_facpub](https://engagedscholarship.csuohio.edu/enece_facpub)

Part of the [Electrical and Computer Engineering Commons](https://engagedscholarship.csuohio.edu/)

How does access to this work benefit you? Let us know!

**Original Citation**

**Repository Citation**
https://engagedscholarship.csuohio.edu/enece_facpub/335

This Article is brought to you for free and open access by the Electrical Engineering & Computer Science Department at EngagedScholarship@CSU. It has been accepted for inclusion in Electrical Engineering & Computer Science Faculty Publications by an authorized administrator of EngagedScholarship@CSU. For more information, please contact library.es@csuohio.edu.
Stackless Multi-Threading for Embedded Systems

William P. McCartney, Member, IEEE and Nigamanth Sridhar, Member, IEEE

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 UnStackedC, a source-to-source transformation that can translate multithreaded programs written in C into stackless continuations. The transformation can support legacy code by not requiring any changes to application code; only the underlying threading library needs modifications. We describe the details of UnStackedC 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, UnStackedC enables a considerable saving of memory used and power consumed, respectively.

Index Terms—User-level threads, preemptive multi-threading, event-driven execution, embedded systems

1 INTRODUCTION

In this paper, we focus on methods of programming concurrent software running in embedded systems contexts. Embedded systems present many challenges to developers. This research focuses on microcontroller-based embedded systems, since there are billions of these devices sold every year. One of the biggest challenges is memory usage. In personal computers the program and the data both exist in random-access memory (RAM). In a microcontroller the program typically executes from read-only memory (ROM) and the data exists in RAM. Microcontrollers have 4 to 32 times the amount of ROM as RAM [1]. This asymmetry means that microcontroller RAM usage comes at a premium. Microcontrollers usually have on the order of a few kilobytes of RAM for storage [2], [3], although the range is actually quite a bit larger from 10 bytes to several hundred kilobytes.

Multi-threaded programming and event-driven programming are popular approaches for building embedded concurrent systems. While there are several instances in the literature where the two approaches have been presented as opposing forces, Adya et al. [4] provide a nice treatment of identifying the essential differences. The most important difference, it turns out, is the way the execution context stacks are managed: manually in the case of event-driven programming, and automatically (by the compiler) in the case of threaded programming.

The two approaches have distinct advantages and disadvantages. Event-driven programs tend to be very efficient in terms of memory footprint. 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. However, writing event-driven programs is hard: the programmer has to manually rip the stack, and maintain state across multiple tasks. Multi-threaded programs, by contrast, are easier to write and comprehend. The program is expressed as a sequence of actions, without regard for implicit points where one task may yield to another. The stack is managed automatically by the thread scheduler. Several multi-threading solutions have been proposed for embedded systems [5], [6], [7]. The big disadvantage with this approach 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.

There are two main classifications of threads: preemptive and cooperative. Cooperative threads only yield the processor to other competing threads at well-defined points in the execution (e.g., when executing some I/O or other blocking operation). Preemptive threads can be swapped out for another thread at any time. One of the most useful features of preemptive threads is the ability for the system to keep running in case of a thread deadlocking. In a cooperative threading system, on the other hand, if a single thread deadlocks or merely runs slowly, it can deadlock or slow down the rest of the threads in the system.

While the memory overhead of threads may not be a big problem for desktop-style applications, the overhead does become a significant handicap when implementing software for embedded systems (where memory is a scarce resource). As it stands, if the amount of available memory is insufficient for the threads that an application will need, the available alternative is to program the system in an event-driven style. Event-driven programs are hard to write, and even harder to read and reason about [8]. It would be nice to be able to program in a threaded fashion, where the threads did not incur the memory overhead.
In this paper, we present the design and implementation of **UnStacked C**: a source-to-source translator that enables C programmers to write code in a multi-threaded fashion, which then executes using event-driven semantics. The translator converts regular C code that uses threads into stackless continuations. Since these continuations 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 threaded fashion and then have them execute in an event-driven manner.

With **UnStacked C**, a large class of 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 TOStThreads API [6] allows us to recompile TinyOS applications written using TOStThreads with almost no changes to application code.

We make the following contributions in this paper:

1. A translation strategy for translating code written using a threads library into stackless continuations.
2. An implementation of this translation for sensor network applications written using the TOStThreads library.
3. A detailed evaluation of this translation in the context of a number of TinyOS applications.

The rest of the paper is organized as follows. In Section 3, we describe the strategy we use for translating threaded C code into **UnStacked C** code. Section 4 presents details on how the source-to-source translator works. We present some analysis of our **UnStacked C** implementation in Section 5, and present results from performance evaluation when comparing **UnStacked C** to TOStThreads in the TinyOS context in Section 6. In Section 2, we present an overview of related work. We conclude in Section 7.

## 2 RELATED WORK

A number of articles have been published in the area of concurrent programming [8], [9], [10], [11], [12], and in particular, focused on the debate between event-driven and multi-threaded programming [4], [13], [14]. In this section, we will review some of the salient pieces of work as they relate to the ideas we present in this paper.

An abbreviated presentation of the evaluation of **UnStacked C** can be found in [15]. This paper describes the translation of threaded code into event semantics in greater detail, and provides a more comprehensive treatment of the subject.

### 2.1 Small Embedded Systems

The nesC programming language [16] 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 [3] is extremely efficient, and is very attractive given the resource-constrained hardware platforms that are targets for these programs. (The MicaZ platform has 4 K of RAM, and the TelosB platform has 10 K.) In our comparison, we can place nesC at the bottom-right of a spectrum: very memory-efficient, but poor in program expressiveness.

At the diametrically opposite corner lie threading approaches with fixed-size stacks such as TOStThreads [6], TinyThreads [5], and MANTIS [7]. 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.

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.

Fig. 1 shows several different concurrent programming models compared with **UnStacked C**. Not only does **UnStacked C** support a reduced RAM footprint, it does so without compromising features that programmers are accustomed to. This includes data integrity. Anytime 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 a lazy preemption technique 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 4.1.

Protothreads [17] in the Contiki operating system [18] provide an alternative in terms of memory efficiency—each thread only requires 2 bytes of memory. However, protothreads are not as expressive as regular threads: they do not support automatic local variables, and state shared across multiple threads must be stored globally. In spite of this disadvantage, the big advantage of protothreads over other

---

1. Section 4.3 presents some limitations of **UnStacked C**, and describe the kinds of applications that will, in fact, require refactoring before the translation can be applied.
threading solutions for embedded systems is extremely small memory footprint. UnStacked C uses a similar implementation technique as that used by protothreads.

Ceu [19] is a synchronous system-level programming language that is designed with concurrency safety as a primary abstraction. The core contribution of this language is that concurrency bugs can be identified and dispatched statically at compile-time, and do not threaten an application at runtime. There are classes of applications that can clearly not be implemented using Ceu (anything using dynamic support, for example), but the remainder of applications that can be leverage the language are quite rich.

There are other attempts at compiler-centric stackless threading for sensornets, namely Threads2Events [20], Ocram [21], and TinyVT [22]. Threads2Events performs a transform that is similar to UnStacked C in that it create a context to store the local variables and child calls. Instead of passing a reference to the context 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. Ocram is a significant extension to Threads2Events, and provides a “comprehensive compiler-assisted cooperative threading abstraction”. The compiler transformation is flexible, and provides many benefits that are similar to UnStacked C.

TinyVT [22] is another TinoOS centric thread to event compiler which adds cooperative threading. TinyVT is designed to allow users to write their TinoOS 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 [5] is a full-functional threading API for TinoOS 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.

TOSThreads [6] is the de facto standard threading model for TinoOS. It is a preemptive threading library for TinoOS. It is included with TinoOS 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 [23] 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 [24] 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.

2.2 High-Concurrency Servers

Adya et al. [4] discuss the essential differences between event-driven programming and threaded programming. They clarify the distinction in terms of how tasks are managed and how the stack is managed. Event-driven programming involves cooperative task management and manual stack management, while typical (preemptive) threaded programming involves preemptive task management and automatic stack management. They present a system of cooperatively scheduling tasks (fibers in Windows), while managing the stack automatically.

Capriccio [8], [10] is a system that completely eschews the event-driven paradigm, and instead adopts the position that support for multi-threaded programming can be more efficient. This system uses a method of managing thread stacks such that the space allocated to thread stacks is used in an efficient manner. The linked stacks that Capriccio uses are based on the observation that in the common case, most threads only use a small portion of the stacks allocated. Similar to UnStacked C, Capriccio uses a C to C translation to modify existing programs. In contrast with UnStacked C, Capriccio does not perform a transform on the storage of stacks, instead it adds a small amount of assembly language to switch to the next element of the linked stack. The stacks are managed such that they can dynamically grow and shrink depending on runtime needs of individual threads. Capriccio is scalable to 100,000 threads in an application.

The staged event-driven architecture (SEDA) [13] takes an opposing view in that threads are hidden from applications. Instead, services are decomposed into stages, each of which contains a thread pool. Stages are non-blocking and are event-driven. Control transfer from one stage to another is managed using a queue, which serves as an execution boundary. The stages are designed to be self-contained modules with little data sharing across stages. This makes reasoning about SEDA behavior simple. Programmers do have to deal with learning to program in the event-driven paradigm.

Tame [14] is a system that enables programmers to write event-based programs in C++ without having to worry about stack ripping. The Tame system provides a set of primitives in libraries that allows programs to be written as though they were using threads. As such, this work is quite similar to ours in that the Tame primitives result in code that looks similar to UnStacked C code. The Tame primitives translate what look like blocking method calls to
simple event-driven continuations. The big difference between Tame and UnStacked C, is that our system does not require the programmer to use new syntax: existing threaded C code can be recompiled into UnStacked C.

Stackless Python [25] is a modified version of python which does not store any state information on the C stack. Stackless Python adds a type of microthreads which are usually stackless and scheduled cooperatively in terms of a C stack. Newer versions of stackless python add some additional functionalities to support preemption and using the C stack only when necessary.

3 UnStacked C Translation Strategy

The core contribution of UnStacked C is a source-to-source translation that takes as input C code using preemptive threads written using a thread library, and produces as output code that uses stackless continuations. The continuation-based code is semantically equivalent to the original code, but because individual thread stacks are no longer necessary, has a much smaller memory footprint than the threaded code. The basic implementation strategy for the continuations in UnStacked C is based on Duff’s Device [26] (a creative way of expressing unrolled loops in C by taking advantage of fall through in case statements) and reentrant functions. This strategy is similar to the strategy used by protothreads [17] and Tame [14].

3.1 Translation Rules

Here, we describe the rules that we use for translating threaded code into stackless continuations. Throughout this section, we will use the code example in Fig. 2 to illustrate the rules we describe.

3.1.1 Function Signatures

Every function in the input program is modified in two ways to support stackless continuations. First, the function signature is modified to return an integer value that codifies the resulting state of the routine (Line 28 in
Fig. 2). Specifically, the state of a routine is used to properly determine where that routine continues at a later point in the program’s execution. It is worth stressing here that one of three values can be returned: 0 indicates that the function returned normally, 1 that it or a child routine blocked, or 2 that it was preempted. This return value does not capture either the thread state or the re-entry point in the function. Second, in addition to the entry point, each function also needs to “remember” where it left off in terms of its context. In order to provide this, we modify the signature of each function to replace its argument list with a single new argument (Line 28 in Fig. 2). This new argument is a pointer to a structure that contains all of its context. The context includes the current state of the function. In addition, the context also includes the values of all of its local variables, and the context(s) of any blocking child function call(s).

### 3.1.2 Context for Blocking Calls

For every blocking function, a structure is generated that can maintain its context (Lines 15-25). The context structure stores the following information about the function while it is blocked:

- **Current state of the function** (Line 16). This state value is used to determine the entry point of the function upon continuation. The variable is the input to Duff’s device.
- **Arguments to the function** (Line 17). Since the function call is no longer guaranteed to be made exactly once (re-entrancy because of blocking child calls), the arguments to each function are maintained in the context structure within the function. This way, regardless of the number of times a method is invoked, the arguments need never be stored on the system stack.
- **Non-static local variables** (Line 18). Since blocking functions are translated into continuations, the values of local variables must be stored for the duration that the function is blocked waiting on a child blocking call.
- **Contexts of all blocking child function calls** (Line 19-24). For each blocking child function call that a parent function makes, the parent maintains a separate context. These child contexts are all part of the parent function’s context. In order to optimize the storage space required for these child contexts, we store them in a union (since only one blocking child function can be active at any time anyway).
- **Return value.** Since the signature of the function has been modified to return the state of the function, we now store the return value in the context.

### 3.1.3 Blocking Function Calls

When translated from multi-threaded code to event-driven state machines, blocking function calls have to be transformed into non-blocking (split-phase) calls. In essence the translation “rips the stack” automatically. For example, consider the blocking call to send() on line 5 in the threaded version of run(). This call is translated in UnStacked C through a series of steps:

1. The state of the current function (caller) is changed to mark the current location (Line 40), and a label is placed in the code prior to the blocking call (Line 42). This label will enable execution to resume from this point the next time this function continues.
2. The state of the child blocking call is set to 0 (Line 39).
3. The arguments to be sent to the child function are populated in the context structure (Lines 35-37).
4. The call to the child function is made, with a reference to the portion of the context structure that corresponds to this child function as argument (Line 43). If the call to the child function returns any value other than 0 (meaning that the child function is blocked), then the calling function will block as well and return 1 at this point (Line 45). On the other hand, if the child function does not return a blocking state, then the current function will proceed with execution.

### 3.1.4 Preemption

Traditional preemption is implemented inside of an interrupt to force a stack swap. Since interrupts can occur at any time, this can leave variables in unknown conditions (half of the bytes written, etc.). This is one of the reasons that cooperative threading can be safer in terms of data-fault prevention, than preemptive threads [5]. Preemption occurs when a thread needs to stop executing typically because some time has elapsed.

In UnStacked C we use a lazy preemption technique. We still utilize the interrupt to notify preemption must occur. Instead of swapping stacks (since we do not have separate stacks) we set a flag notifying the thread to terminate. This flag is then checked each time a loop iterates. If the flag is set, it will exit the function, returning 2 which notifies the scheduler that it was preempted.

This lazy preemption provides the same data integrity as cooperative threads with the ability to provide preemption. Every variable write will complete prior to preemption occurring. This allows swapping threads without swapping stacks, and the observable functional behavior of the code is the same as that of the event-driven code. There is something to note here, however. The introduction of lazy preemption does “soften” the timing guarantees that the translated code can promise. If timing guarantees are important to the application, the developer will need to pay attention to introducing a flag variable that will force preemption to occur immediately.

### 3.1.5 Order of Operations

When ripping the stack the way we do, one very important thing to consider is how that affects the order in which operations are executed. This problem manifests itself in several different scenarios. Consider the following assignment statement,

```c
i = foo(j);
```

This assignment statement has multiple actions in the same statement. First, the function `foo()` needs to be evaluated, and then the return value is assigned to `i`. If `foo()` is a blocking operation, there needs to be special attention paid to this line. The blocking operation `foo()` needs to be made
in accordance with the translation rule above. Consequently this code is translated as follows:

1. `ctx->children.foo.ops.args.x = j;`
2. `ctx->children.foo.state = 0;`
3. `ctx->state = 1;`
4. `case 1:
   5. if (foo(&ctx->children.foo) != 0)
   6. return 1;
   7. i = ctx->children.foo.ops retval;`

Visualize this in terms of the abstract syntax tree (Fig. 3a) traversal required for this translation to occur correctly. It is not possible to figure out that `foo()` is a blocking operation at the point of traversing the `assn-stmt` node in the AST. It is only when the traversal has moved down past this node, and into the `function-call` node for `foo()` that this determination can be made. At this point therefore, the traversal has to back-track to the assignment statement and split this single statement into a compound statement that looks like the AST presented in Fig. 3b. In essence, when a blocking operation is one of many expressions occurring in a single statement, that blocking operation needs to be stripped off to precede the remaining statements.

There is more. Consider the following statement,

```
1 i = foo(j) + bar(k);
```

where `foo()` is a blocking operation. This code, when translated to `UnStacked C`, becomes the following (just like in the previous example):

```
1 ctx->children.foo.ops.args.x = j;
2 ctx->children.foo.state = 0;
3 ctx->state = 1;
4 case 1:
5   if (foo(&ctx->children.foo) != 0)
6     return 1;
7   i = ctx->children.foo.ops retval +
8     ctx->children.bar.ops retval;
```

However, what if `foo()` was non-blocking, but `bar()` was blocking? Simply applying the rule of moving `bar()` before the rest of the statement is not going to work: the execution of `foo()` now will come after `bar()`, violating the order of operations in the original program. The execution of `foo()` in this case, therefore, the correct translation should be as follows:

```
1 ctx->children.foo.ops retval = foo(j);
2 ctx->children.bar.ops retval = k;
3 ctx->children.bar.state = 0;
4 ctx->state = 1;
5 case 1:
6   if (bar(&ctx->children.bar) != 0)
7     return 1;
8   i = ctx->fooretval +
9     ctx->children.bar.ops retval;
```

This also means that in such cases, the contexts of child function calls cannot be stored in a union, since the return values of multiple functions must be stored. In order to maintain order of operations in this case, the contexts must be stored as a nested struct.

### 3.1.6 Variables

A key feature of `UnStacked C` is that the system stack is not used for maintaining local variables in blocking operations. Instead these variables are maintained in the contexts of these blocking operations. Neither global variables nor static variables are touched in this translation. The AST of the program is traversed to transfer the local variables from blocking functions into context structures that correspond to each blocking function.

While the declarations of local variables are transferred to context structures, the initialization statements are retained in the same place, and only names of the variables are changed:

```
1 int i = 7;
```

is translated to:

```
1 ctx->locals.i = 7
```

Here again, order of operations matters. If the right hand side of the assignment is a blocking function call, then that blocking function is moved above the assignment.

### 3.2 Correctness of Translation

The translation we perform is achieved by a complete unrolling of the AST using whole program analysis. The
The core of the translator [27]) is built on top of the C Intermediate Language (CIL) toolset [28]. Transformations that CIL produces have been shown to be correct, and we can leverage those correctness guarantees. We have implemented a number of applications in the TinyOS distribution using the TOSThreads library, and compared the behavior the native implementation of every application against the `UnStacked C` implementation to ensure that the observable behavior is properly maintained.

4 TOSTHREADS IMPLEMENTATION

`UnStacked C` uses a C-to-C translator called C-XML-C [27] to translate multi-threaded code into stackless threads. Our implementation of this translation is built using the CIL translation framework [28]. While the entire transform could have been implemented in CIL directly, we also relied on patches to serialize the data prior to our processing [29]. 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 tree modifications to produce a resultant tree as described in Section 3.

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 stack-swapping routines will need to be replaced with simpler routines. Another major kind of modifications involve 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.

4.1 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 Fig. 6.
an interrupt fires, if a task has been posted, it tries to wake up the TinyOS thread (if it is not already active) and to switch to that thread. The original TOSThreads implementation of `interruptCurrentThread` can be found in Fig. 7 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 significantly simpler. Preemption in the UnStacked C implementation of TOSThreads is done in a “lazy fashion”. In a traditional preemptive threading system (such as TOSThreads), when an interrupt occurs, the current thread is immediately preempted, and the interrupt is handled. By contrast, in cooperative threading systems (such as TinyThread), threads run to completion, and can only be stopped at pre-defined points. Our lazy preemption scheme sits in between the two schemes listed above. 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 gotos which branch backwards. These test points are added by the translator automatically without any developer input.

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 no longer near infinite—it 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 is shown in Fig. 8. Since we are using lazy preemption, there is no harm in calling `interruptCurrentThread()` excessively, so several of these checks do not need to be in place. The UnStacked C code for `interruptCurrentThread()` can be found in Fig. 9 and it simply sets a preemption flag to be checked inside of the threads themselves.

Preemption among threads in TOSThreads is handled by a similar mechanism. The software timers in TinyOS are utilized to implement the `PreemptionAlarm` which forces a swap between threads. The original code is shown in Fig. 10. 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 Fig. 11.

### 4.2 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.

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 Fig. 12. Instead of this, in UnStacked C a context is allocated for each instance. This context is calculated by the compiler after it compiles 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. If the right amount of memory is not allocated, there can be adverse results. The “safe” option is to over-estimate the amount of memory needed (as is done with [7]), but this can result in wasted resources. If the allocation tends to be aggressive, and underestimates, the program will crash. A detailed treatment of stack allocation is presented in [30] and [5].

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 the 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 Fig. 13.

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.

4.3 Limitations

The current version of UnStacked C has three main limitations. 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.

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. It would be easy to add a spawn routine to the compiler to support such an operation, but allocation of the required context would have to be done by the developer. 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 last 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.

All of these limitations only impact the functions which are marked as blocking. If the rest of the program contains any of these they will not be impacted, only the blocking threaded code.

5 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. Fig. 14 shows a generic call graph. Each node represents a function call. The total stack depth the number of function calls is not determined solely by the number of calls, but by the sum of the function call overheads summed to the maximum possible amount.

Fig. 14 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 $\text{Main}(5) + a(6) + b(8) + \text{Foo}(12) + \text{Bar}(8) = 39$.

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. Fig. 15 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 Fig. 16 which gives a greater maximum stack depth.

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

Fig. 12. Original TOSThread stack allocation code where stack_size is a parameter that the developer estimates as the stack size.

```
1 //The code the programmer writes
2 #define STACKSIZE(NAME)
3 int __attribute__((blockingstack(NAME)))
4 STACKSIZE(run_thread) threadstack;
5 //The code after translation
6 struct UnStacked_run_thread threadstack;
```

Fig. 13. Updated UnStacked C context allocation code, where the compiler calculates the exact required RAM for a function named run_thread.
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 a contiguous fashion.

A function’s stack overhead is made up of two main parts, 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 Fig. 17. It is important to notice how each thread requires an overhead for the interrupts.

Fig. 18 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 in the global variables. In many event driven systems, programmers have optimized the system by not storing the equivalents of automatic variables globally if they are not used across multiple events.

In our transforms instead of storing the program counter we store a state index, but similarly to threaded stack we still store the automatic variables. This is shown in Fig. 19. Notice that in both the stackless (Fig. 19) and the event driven (Fig. 18) RAM usage by each of the threads can be reduced significantly.

6 PERFORMANCE EVALUATION

In order to evaluate the performance of UnStacked C threads, we ran two sets of experiments. In the first set of experiments, we compare the performance of UnStacked C threads with the performance of multi-threaded wireless sensor network applications written for the TinyOS platform using the TOSThreads library. In addition to this comparison, our second set of experiments is based on a microbenchmark, which is intended purely as a proof-of-concept to demonstrate UnStacked C’s potential as a scalable solution to minimizing the amount of memory consumed by thread stacks. Our microbenchmark results show that using UnStacked C, programs can use a number of threads that is not possible using traditional threads libraries.

6.1 TinyOS Applications

TOSThreads [6] is a threads library that allows developers to write multi-threaded applications for the TinyOS platform. When using this library, developers enjoy the convenience of programming procedural code, and yet enjoy the performance benefits of event-based execution at the lower levels of the operating system. TOSThreads does not even require the developer to identify explicit points of synchronization (as in cooperative threads). Instead, the library supports full preemption.

For our evaluation, we used seven applications that are in the default TOSThreads distribution [31]; we did not implement any new applications for this comparison so as not to bias the 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 process workflow is a different invocation to the build system. The tests were all conducted on the same compiling host with the TinyOS 2.1.0 distribution and GCC 3.2.3.

6.1.1 Memory Usage

Fig. 20 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 percent. 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, Fig. 20 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 percent.

This reduction in RAM usage comes at a cost, however, since additional code is generated. The ROM (flash) comparison is shown in Fig. 21. The overhead of the ROM is only a 12 percent 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 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 store the values in an orderly fashion.

One may note that in absolute terms, the use of UnStacked C actually represents an increase in total memory used (RAM + ROM), and this is, in fact, true. However, in small embedded systems, the amount of RAM available is significantly smaller than the amount of ROM. In a TelosB mote, for example, the amount of RAM is 10 KB, whereas the flash size is 1 MB. So it is important to keep these two measures of memory (RAM and ROM) separate from each other. In this context, a 100 byte savings in RAM is much more valuable than a 1 KB overage in ROM.

6.1.2 Power Consumption

For most sensing applications, the CPU on the sensor node goes to sleep 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 interrupt overhead, 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

---

**Fig. 20. TinyOS RAM usage in TOSThreads versus UnStacked C.**

**Fig. 21. TinyOS Flash usage in TOSThreads versus UnStacked C.**

**Fig. 22. CPU current consumption of Blink in TOSThreads versus UnStacked C.**
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 modifications at the application code level. The applications are simply recomplied with the new implementation of the threads library in order to enjoy the cost savings provided by UnStacked C.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation through Grant CNS-0746632. A shorter version of this paper appeared in the Proceedings of the 2011 International Conference on Distributed Computing on Sensor Systems (DCOSS11). Nigamth Sridhar is the corresponding author.

REFERENCES


2. All source code for UnStacked C along with the test data are available for download at http://www.unstackedc.com.