# Poster Abstract: Compiler-Assisted Thread Abstractions for Resource-Constrained Systems

Alexander Bernauer
Institute for Pervasive Computing
ETH Zurich
bernauer@inf.ethz.ch

Kay Römer
Institute of Computer Engineering
Universität zu Lübeck
roemer@iti.uni-luebeck.de

***Abstract*—Major operating systems for wireless sensor networks (WSN) enforce an event-based programming paradigm for efficiency reasons. However, practice has shown that the resulting code complexity leads to problems during development, deployment, and operations. Although thread-based programming is known to solve these problems, the scarce resources of common WSN devices make it non-trivial to actually support it.**

**As opposed to existing runtime-based thread libraries, our goal is to explore the potential of compiler-assisted thread abstractions by introducing a comprehensive and platform-agnostic system of compiler and debugger which supports cooperative threads with minor restrictions.**

**The compiler allows to write thread-based programs that are automatically translated to equivalent event-based programs, while the debugger provides source-level debugging of the initial program, thus sustaining the thread abstraction.**

**Our preliminary results and ongoing evaluations suggest that the resource-wise overhead of the abstraction is moderate and can be below the overhead of runtime-based solutions. We also demonstrate that the transformation is platform-agnostic by supporting both Contiki and TinyOS.**

## I. Introduction

Major WSN operating systems (OS) such as TinyOS [4] and Contiki [2] account for the scarce resources of WSN devices by offering asynchronous application programming interfaces (API) and by imposing the event-based programming paradigm to its applications.

Although efficient, practice has shown that the implications of this paradigm often pose significant problems to developers. This is in particular true for the WSN domain, as deployment environments tend to differ strongly from lab environments and debugging of deployed networks is usually very time- and energy-consuming. Therefore, mistakes resulting from the inherent complexity of event-based programming tend to be very expensive to cope with.

In order to eliminate this source of mistakes altogether, researchers have investigated the question how to support thread-based programming on WSN devices despite the scarce resources. The prevalent system in the Contiki world is Protothreads [3] where a set of C preprocessor macros enable the syntactical illusion of threads and synchronous operations. In contrast, the most wide-spread approaches for TinyOS are runtime-based thread libraries such as TOSThreads [5], but with TinyVT [6] there is also a translation-based solution. There, a dedicated compiler generates a component's implementation from sequential nesC code which is enriched with special `await` statements where the control flow blocks until the occurrence of an event.

As opposed to runtime-based solutions, compilers can exploit application-specific properties and apply optimizations. This is why we hypothesize that compiler-assisted thread abstractions can be more resource-efficient than thread libraries. Additionally, we argue that the provided thread abstraction should also be sustained during debugging. None of the existing thread abstractions achieve this, though, and existing compiler-assisted thread abstractions have significant limitations concerning the supported thread semantics.

Thus, our goal is to take the next step of compiler-assisted thread abstractions. We aim to verify our claims by presenting a platform-agnostic source-to-source translation scheme. This scheme translates ISO/IEC 9899 (C99) applications using cooperative threads with synchronous OS APIs into C99 applications which use events with asynchronous OS APIs while preserving the operational semantics.

In preliminary work we have evaluated an initial translation scheme using a worst-case application [1]. Given the fact that we have considerably improved the translation scheme since then and there are still many opportunities for optimizations that we are going to exploit, we are confident that the efficiency of generated applications can be very close to the efficiency of hand-written applications. In the following we sketch this new translation scheme.

## II. Transformation

The input to the compiler is an OS API specification consisting of declarations of synchronous functions, and thread-based application code that uses this API (T-code). In this context, every synchronous API function is a so-called *critical function* and every function that calls a critical function, i.e. contains a *critical call*, is also critical. Non-critical functions are not altered by the transformation.

Concerning the critical functions, there are some limitations to the supported thread-semantics due to the fact that compilation is restricted to decidable problems. Thus, it is forbidden a) for critical functions to be recursive, b) to call critical functions via function pointers, and c) to perform pointer arithmetics that escapes the memory location of an object. Additionally, the number of threads is a compile-time constant.

The input OS API is translated to an asynchronous API and the T-code is translated to an equivalent event-based application that uses this API (E-code). To actually execute E-code, it is necessary to provide a *platform abstraction layer* (PAL) that implements the E-code API by using the existing API of the employed OS to trigger the desired operations and register the corresponding callbacks.

Both T-code and E-code are non-deterministic programs. We thus define an E-code to be equivalent to a T-code if and only if every possible execution of the E-code has the same *observable behavior* as at least one possible execution of the T-code. Hereby, the observable behavior of a program is the sequence of all API calls including all input parameters. Note that not including the exact timing of API calls imposes no additional restrictions, as cooperative threads are not viable for timing-critical applications anyway [5].

The transformation is sound and complete regarding the equivalence of T-code and E-code, because translating the control flow preserves the sequence of language statements while translating the data flow preserves their individual effect. We achieve this by the following means:

Concerning the data flow, for every critical function, a C structure that we call *T-frame* is generated. It stores the function's local variables, its parameters, its return value when appropriate, the caller's continuation and a union of all T-frames of its callees. For every *thread starting function*, i.e. a critical function without callers, its T-frame is instantiated once which constitutes the *T-stack* of this thread. By design, a T-stack simulates the runtime stack of a T-code thread if it would be actually executed. Thus, the translation can replace all read and write accesses to local variables with read and write accesses to T-stack variables. Furthermore, T-stacks can help managing the control flow as follows.

For every thread starting function, a so-called *thread execution function* is generated that comprises the inlined bodies of all critical - but not blocking - functions that are directly or indirectly invoked by the thread starting function. Every call to a critical function is then rewritten to the following sequence: First, the function parameters are written to the T-stack. Then, the continuation, which is the address of the label[1] preceding the first statement after the call, is written to the T-stack. Last, the control flow jumps to the label that precedes the callee's function body. Similarly, returning from a critical function results to writing the function result to the T-stack and jumping to the continuation as stored on the T-stack. Then, the caller can retrieve the result of the function call from the T-stack and continue its computation.

Invoking blocking functions also involves writing the function parameters and the continuation to the T-stack. However, the next step is to trigger the desired operation by calling the PAL's implementation of the blocking function while passing the pointer to its T-frame. As soon as the operation is finished, the PAL's obligation is to invoke the thread execution function and pass it the continuation information as previously saved on the T-stack. Then the thread execution function can jump to the continuation, fetch the operation's results from the T-stack and continue its computation.

As already mentioned, compilers can exploit application-specific optimizations. For example, if a critical function is only called once in the whole program, the caller's continuation can be hard-coded into the E-code instead of being memorized. Similar optimizations exist for read-only function parameters and automatic variables that are not read after a critical call and thus can stay on the E-code stack.

In either case, the compiler can create a log of all applied transformations which can be used by a T-code debugger to map locations and variables between T-code and E-code. By using a conventional C debugger that monitors the E-code, the T-code debugger can thus provide the well-known concepts of source-level debugging for T-code applications.

[1]Computed gotos is a GNU extension which can easily be recreated by jump tables if standard compliance is important.

## III. Outlook and Conclusions

In order to evaluate the efficiency of our approach, we are planning to implement various WSN applications on top of both Contiki and TinyOS. One variant of each application will use the native event-based OS API and one will use our compiler prototype. Given such applications, we will measure their resource consumption using both static tools and simulators. The interesting metrics are a) the size of the binary, i.e., the ROM consumption, b) the RAM consumption, c) the number of CPU cycles required for one iteration of each recurring application task, and d) the total energy consumption.

Overall, we have shown how compiler-assisted thread abstraction can support almost complete thread semantics in a platform-agnostic manner. Furthermore, we have explained why we expect them to be both more efficient than runtime-based solutions and almost as efficient as hand-written event-based applications.

## References


[1] Alexander Bernauer, Kay Römer, Silvia Santini, and Junyan Ma. Threads2Events: An Automatic Code Generation Approach. In *Proceedings of the 6th Workshop on Hot Topics in Embedded Networked Sensors*, 2010.

[2] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, 2004.

[3] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems. In *Proceedings of the 4th ACM Conference on Embedded Networked Sensor Systems*, pages 29–42, 2006.

[4] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. *SIGPLAN Not.*, 35(11):93–104, 2000.

[5] Kevin Klues, Chieh-Jan Mike Liang, Jeongyeup Paek, Razvan Musaloiu-E, Philip Levis, Andreas Terzis, and Ramesh Govindan. TOSThreads: thread-safe and non-invasive preemption in TinyOS. In *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 127–140, 2009.

[6] János Sallai, Miklós Maróti, and Ákos Lédeczi. A concurrency abstraction for reliable sensor network applications. In *Proceedings of the 12th Monterey conference on Reliable systems on unreliable networked platforms*, pages 143–160, 2007.