



Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Compiler-Assisted Thread Abstractions for Resource-Constrained Systems

Alexander Bernauer

Supervisor: Prof. Dr. Friedemann Mattern

Institute for Pervasive Computing

Department of Computer Science, ETH Zurich

Summary

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 is hard to manage for software developers and leads to difficult problems during development, deployment, and operations. Thus, thread libraries for WSN applications have been introduced, but the efficiency constraints of the domain often lead to glaring restrictions of the supported thread semantics.

In contrast, compiler-assisted thread abstractions support threads without actually using threads at runtime. Instead, the thread-based code is automatically translated into equivalent event-based code. Currently, two different systems implement this idea, but both have severe limitations regarding both the thread semantics and the tool support.

Our goal is to demonstrate the potential of compiler-assisted thread abstractions by taking the next step and introducing a comprehensive system of compiler and debugger which supports cooperative threads with minor restrictions and which is also platform independent.

A Haskell-based compiler prototype shows the feasibility of our approach and preliminary results demonstrate that the generated code is efficient enough to be executed on wireless sensor network devices. Furthermore, existing optimization potential suggests that compiler-assisted thread abstractions can indeed outperform runtime-based solutions and compete with hand-written event-based code. We are currently working on completing the implementation of the tools and verifying these points by means of an extensive evaluation.

1 Introduction

Wireless sensor networks (WSN) aim to provide monitoring of physical phenomena over a spatial region without requiring existing infrastructure. To this end, battery-powered computing systems with sensors and wireless communication capabilities are deployed in the field and continuously deliver measured values via multi-hop routing to a base station.

Examples of existing WSN applications include monitoring of the permafrost in the Swiss Alps [8], adaptive lighting in operational road tunnels [4], and wildlife monitoring [7]. Such applications typically require a long system lifetime without manual maintenance such as renewal of batteries. As a consequence, energy efficiency is one of the most critical design criteria for WSN applications, which in practice implies resource-constrained devices. A typical example of such a device is the T-Mote Sky, which is equipped with a 16 bit, 8 MHz microcontroller, 48 kB of Flash memory and 10 kB of random access memory (RAM).

Major WSN operating systems such as TinyOS [9] and Contiki [5] account for the scarce resources by enforcing an event-based programming paradigm. The common reasoning behind this is that with event-based programming, a single dispatcher thread and a single stack suffice to execute multiple tasks concurrently.

Although efficient, practice has shown that the implications of this paradigm often pose significant problems to developers, as it is difficult and cumbersome to manually manage the states and the control flow of event-based applications [1]. Furthermore, deployment environments typically differ strongly from lab environments and debugging is usually very time- and energy-consuming. Therefore, mistakes resulting from the inherent complexity of event-based applications tend to be very expensive to cope with.

To better support developers in programming WSN applications, investigations have been started to provide thread-based programming for WSN applications. On a first branch, common approaches to thread libraries have been adapted, sometimes by introducing restrictions in the supported thread semantics [10, 11, 13]. On a second branch, compiler-assisted thread abstractions employ a compiler which takes a thread-based program as input and generates an equivalent event-based program as output. By these means software developers have the comfort of threads while the runtime still performs efficient event dispatching.

The goal of this thesis is to investigate the potential of compiler-assisted thread abstractions for resource-constrained systems. In particular, we aim at demonstrating that compiler-based solutions can not only support almost complete thread semantics to the programmer but also outperform runtime-

based solutions in terms of code efficiency. To this end, we are building an accordant compiler called Ocram and will evaluate its performance characteristics on the basis of typical WSN applications. The final outcome of the thesis will thus consist in both a quantitative and a qualitative analysis of the potential and limits of compiler-assisted thread abstractions.

In the next section we will discuss two existing systems that already provide compiler-assisted thread abstractions for WSN applications. Being the first of their kind, though, both systems have severe limitations. Ocram is supposed to overcome these limitations as described in section 3. Section 4 subsequently presents published results, the current state of the work and future plans of it.

2 State-of-the-Art

The first system that provided a compiler-assisted thread-abstraction for WSN applications is Protothreads [6]. It has been specifically designed for Contiki and its usage is deeply embedded in Contiki's libraries and runtime system. Technically, Protothreads are a set of C preprocessor macros that enable the syntactical illusion of threads and blocking operations. Instead of waiting for the completion of a blocking operation, though, the runtime memorizes the current code location and returns from the handler function. When resuming the thread, the same function is called again and a `switch` statement, which is expanded from the Protothreads macros, brings the execution back to the location where the previous blocking operation occurred.

As the C preprocessor can only locally replace language tokens, this approach has a number of limitations. First of all, the state of automatic variables is not preserved across calls of blocking operations. Second, blocking operations can only be called in the topmost thread function, which severely restricts the software architecture of Protothreads-based applications. Last but not least, certain user mistakes are not detected at compile time but have to be indirectly examined by observing unexpected execution behavior. Such mistakes can be as subtle as using additional `switch` statements that interfere with the expanded ones or relying on the state of an automatic variable after a blocking operation.

In contrast, the first system that employs a dedicated compiler to provide thread abstractions was TinyVT [12]. It has been specifically designed for the TinyOS operating system which is implemented in the nesC programming language [9] and enforces a component-based software architecture with event-based interfaces. TinyVT enables software developers to implement single components sequentially as it is possible to inline event handlers in

code blocks following a special `await` statement. Also, the runtime preserves the state of local variables across such operations.

Although TinyVT overcomes many of Protothreads' drawbacks, the supported thread semantics is still rather restricted. First of all, inlined event handlers may not contain `await` statements. Additionally, it is not possible to split the implementation of a component into multiple functions, which implies that code can not be shared between multiple threads, which in turn implies that TinyVT has no support for reentrant code.

Besides the above mentioned limitations, both systems provide no support for debugging. The consequence in practice is that software developers use source-level debuggers to step through the generated event-based code. We consider abstractions without debugging support as incomplete because they sometimes fail to hide lower-level details although their primary purpose is to relieve the user from having to deal with them. Even worse, software developers additionally need to understand the implementation details of the abstraction itself in order to perform the necessary back-mapping to identify the cause of the observed error in the abstract program.

In contrast to Protothreads and TinyVT, Ocrum translates from and to standard-compliant C code, supports thread semantics comparable to what common thread libraries provide, and is platform independent with respect to the underlying event-driven runtime environment. Furthermore, we want to provide a corresponding source-level debugger that works on the level of the thread abstraction and offers a feature set that is comparable to those of common source-level debuggers. Finally, we expect that our findings are transferable to other domains where the event-based programming paradigm is prevalent. The next section presents the details of this system.

3 System Design

The Ocrum compiler translates thread-based code (*T-code*) to event-based code (*E-code*) while both T-code and E-code comply with the ISO/IEC 9899 international standard (C99). Section 3.1 describes the semantics of the T-code and the limitations concerning the set of supported language features. Next, section 3.2 illustrates how Ocrum translates T-code into E-code. The interoperability with the runtime environment that executes the E-code is discussed in section 3.3, which completes the description of the system basics. Subsequently, section 3.4 enumerates a set of optimizations opportunities that can be exploited to generate more efficient E-code. Finally, section 3.5 describes the design of the T-code source-level debugger and the requirements it poses on the compiler.

3.1 Thread Semantics and Limitations

On the T-code side, Ocram supports cooperative threads with shared memory. We assume that the underlying runtime environment defines in its application programming interface (API) the set of blocking functions together with non-blocking functions and required types. Every call to a blocking function is an implicit yield point where control could be passed to a different thread. In addition to that, the runtime environment can support explicit yield points by means of a blocking function that performs no operation.

Every blocking function is called a *critical function* and every function that calls a critical function, i.e. contains a *critical call*, is also a critical function. Every thread is started by the call of its *thread start function*, which usually is a critical function itself. Critical functions are reentrant and can thus be used by different threads. Furthermore, arbitrary non-critical functions can be used to perform additional computations. As threads are not preempted they can easily share memory and thus communicate with each other by means of global variables. In general, T-code is basically just C99 code, which has the additional benefit that existing development environments can be used to write it.

Since the thread semantics are implemented at compiler level, there are some inherent limitations, though [3]. First, the number of threads must be known at compile time. Second, recursive functions must not invoke critical functions. And third, function pointers must not be used to invoke critical functions. We argue, though, that these limitations do not severely affect programming of WSN applications. In particular, recursive algorithms tend to have an undecidable stack consumption and thus should generally be avoided in embedded systems. Again due to the constrained resources of the employed devices, it is often not sensible to support an arbitrary number of threads. Finally, the use of function pointers is not uncommon, but can be avoided by a case differentiation with moderate overhead. In any case, the compiler can reliably detect any violation of these constraints.

3.2 Compilation Scheme

As a first step, the compiler must rewrite the control flow of the program to replace critical calls with non-blocking *split-phase operations* [9]. Additionally, it must rewrite the data flow to preserve the value of automatic variables. Amongst many possibilities, we are currently investigating the following compilation scheme as our preliminary results suggest that it produces efficient E-code in most cases.

For every critical function, the Ocram compiler generates a so-called *T-*

stack structure that holds all automatic variables, all function parameters, the function's return value (if existent), the continuation of the current caller, and an instance of the T-stack structure of each critical function that is directly called by the respective function. Furthermore, one instance of the T-stack structure of every thread start function is generated and used by the E-code to access the corresponding variables. We call such an instance a *T-stack*.

As the scope of a T-stack structure is the execution of the corresponding function and as every instance of a T-stack structure belongs to exactly one thread, all nested T-stacks can share memory and are thus grouped in a C union. In fact, a T-stack represents the overlay of all snapshots of the runtime stack of the corresponding thread if it would actually be executed.

Concerning the control flow, a critical function must only trigger the respective operation and return immediately. The platform abstraction layer, which is covered in section 3.3, implements the blocking functions accordingly. For the other critical functions, the compiler replaces every critical call with the following sequence of statements: First, the callees parameters are written to the T-stack. Second, the continuation information for the callee is written to the T-stack. Third, the actual call is performed. Last, the function returns.

The bodies of all critical functions that are used by a thread are all inlined into one common *thread execution function* for each thread. Thus, the continuation can simply be the address of a label and a call of a critical, but not blocking, function is just a `goto`.¹ The first statement of every inlined body, as well as every statement after a critical call, is equipped with a unique label. Furthermore, every `return` statement in the T-code is replaced by a `goto` statement that uses the continuation information stored on the T-stack.

As there is a one-to-one mapping between thread execution functions and T-stacks, read and write access to the T-stack involves no indirections. The C compiler that processes the E-code can thus generate efficient memory access code despite the nested structure of the T-stacks.

3.3 Platform Abstraction Layer

The above described compilation scheme depends on proper event-based implementations of the blocking functions that are used by an application. Their role is to implement the systematic interface assumed by the code generation by means of the interface of the respective underlying runtime

¹Computed gotos are a GNU extension to the C programming language. If it is important to stay standard compliant they can be emulated via less efficient jump tables.

environment. This manual effort is necessary because existing runtime environments have non-systematic interfaces so that the code generation needs some sort of user assistance to be able to use them. Amongst many possible ways of formalizing those interfaces we think that simply implementing them once is justifiable and feasible.

Clearly, this platform abstraction layer (PAL) introduces an overhead in terms of code size, RAM usage and CPU cycles which we have to account for when evaluating the overall system performance. The size of this overhead inherently depends on the semantics of the underlying interfaces. Given a runtime system that actually support Ocrum natively this overhead could even disappear.

As the Ocrum compiler poses little requirements on the PAL, it is in general not restricted to the WSN domain. In any other domain, where the event-based paradigm is prevalent for efficiency reasons and C99 is used, software developers could benefit from our compiler-assisted thread abstraction.

3.4 Optimizations

The compilation scheme of section 3.2 supports the most generic case that can be encountered. In practice, though, many typical recurring patterns exist and allow for shortcuts that can be exploited by a compiler.

First of all, automatic variables that are never written to before a critical call and read afterwards do not need to be preserved, so they do not have to be saved on the T-stack. Instead, they can stay automatic variables in the E-code as well and, thus, be saved on the single runtime stack which saves memory and access time.

As a second example, if a function is only called from one location in the whole program there is no need to save the continuation in memory. Instead, it can be hard-coded into the E-code thus saving memory and lookup time. The callee can additionally access the variables of the caller in case of read-only parameters.

We believe that further investigations will reveal additional potential for optimizations that all sum up to a significant performance benefit of compiler-assisted thread abstractions compared to runtime-only solutions.

3.5 Debugging Support

The role of the debugger is to undo the compilation so that the E-code runtime can be presented by means of the T-code abstraction. In general this involves managing the mapping of both the data and the control flow. To this end, the T-code debugger utilizes and controls an E-code debugger

which is just a standard C debugger, possibly attached to the execution via JTAG² or other platform dependent means.

When the user wants to know the value of a T-code variable the T-code debugger only needs to know which E-code variable holds this value and request it from the E-code debugger. Similarly, when the user installs breakpoints or issues step-through commands, the T-code debugger needs to know to which location in the E-code this corresponds to and issue the proper commands to the E-code debugger.

To enable the debugger to perform these tasks, the Ocram compiler must generate corresponding debugging information for all variables and all statements of critical functions. This relatively simple technique is enough to add significant value to the thread abstraction provided by our Ocram compiler as discussed in section 2.

4 Past, Current, and Future Work

In [2] and [3] we have presented a first compilation scheme that differs from the one described in section 3.2. Since then we have been working both on finding better compilation schemes and implementing the Ocram compiler.

For the latter, we eventually have decided to use the Haskell programming language, for which the Language.C module provides a comprehensive parser and pretty-printer for the C programming language. The current version of the Ocram compiler³ is a proof-of-concept implementation. Although it supports the translation of basic applications, many C language features are not supported yet. Furthermore, we have implemented a basic PAL for both Contiki and TinyOS and successfully executed generated E-code on a T-Mote Sky device.

As a next step, we plan to implement the T-code debugger and to extend the feature set of Ocram and exploit possible optimizations to achieve higher efficiency. Our goal is to outperform existing threading libraries in an evaluation involving realistic WSN applications. Finally, past work has shown that the efficiency of the compilation scheme highly depends on application properties such as the number of reentrant functions. On the long term we thus envision a system that measures relevant application properties and chooses the best compilation scheme accordingly, possibly involving user interaction to achieve a proper trade off.

²IEEE 1149.1

³<http://github.com/copton/ocram>

References

- [1] Atul Adya, Jon Howell, Marvin Theimer, William J. Bolosky, and John R. Douceur. Cooperative Task Management Without Manual Stack Management. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 289–302, 2002.
- [2] Alexander Bernauer, Kay Römer, and Silvia Santini. Threads for the Programmer, Events for the Machine. In *Adjunct Proceedings of the 7th European Conference on Wireless Sensor Networks*, pages 2–4, 2010.
- [3] 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.
- [4] Matteo Ceriotti, Michele Corrà, Leandro D’Orazio, Roberto Doriguzzi, Daniele Facchin, Stefan Gun, Gian Paolo Jesi, Renato Lo Cigno, Luca Mottola, Amy L. Murphy, Massimo Pescalli, Gian Pietro Picco, Denis Pregolato, and Carloalberto Torghele. Is There Light at the Ends of the Tunnel? Wireless Sensor Networks for Adaptive Lighting in Road Tunnels. In *Proceedings of the International Conference on Information Processing in Sensor Networks*, pages 187–198, 2011.
- [5] 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.
- [6] 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.
- [7] Vladimir Dyo, A. Stephen Ellwood, David W. Macdonald, Andrew Markham, Cecilia Mascolo, Bence Pasztor, Salvatore Scellato, Niki Trigoni, Ricklef Wohlers, and Khar-sim Yousef. Evolution and Sustainability of a Wildlife Monitoring Sensor Network. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, pages 127–140, 2010.
- [8] Andreas Hasler, Igor Talzi, Jan Beutel, Christian Tschudin, and Stephan Gruber. Wireless Sensor Networks in Permafrost Research Concept , Requirements , Implementation and Challenges. In *Proceedings of the 9th International Conference on Permafrost*, pages 669–674, 2008.
- [9] 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.
- [10] William P. McCartney and Nigamanth Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In *Proceedings of the 4th ACM Conference on Embedded Networked Sensor Systems*, pages 167–180, 2006.
- [11] Christopher Nitta, Raju Pandey, and Yann Ramin. Y-threads: Supporting concurrency in wireless sensor networks. *Distributed Computing in Sensor Systems*, pages 169–184, 2006.

- [12] Janos Sallai. *Compiler-assisted concurrency abstraction for resource-constrained embedded devices*. Dissertation, Vanderbilt University, 2008.
- [13] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, pages 29–42, 2004.

5 Biography

Alexander Bernauer graduated in Computer Science from the University of Ulm, Germany in 2006. Subsequently he was been working as a C++ software engineer developing tracking and navigation products at a start-up company in Heerbrugg, Switzerland. Since 2009 he has been a Ph.D. candidate under the supervision of Prof. Friedemann Mattern at the Institute of Pervasive Computing, ETH Zürich, Switzerland. His expected date of dissertation is February 2013.