Poster Abstract: Threads for the Programmer, Events for the Machine

Alexander Bernauer[1], Kay Römer[1,2], and Silvia Santini[1]

[1] Institute for Pervasive Computing, ETH Zurich, Switzerland
[2] Institute of Computer Engineering, University of Lübeck, Germany
{bernauer,roemer,santinis}@inf.ethz.ch

1 Introduction

Major motes operating systems like TinyOS or Contiki [1, 2] rely on an event-driven programming paradigm. While the use of events allows for limiting memory usage on resource-constrained motes, it may also hamper the development and debugging of applications, especially as their complexity increases [3]. Several authors also investigated the possibility of introducing threads to mote programming [4–6]. However, the proposed solutions all induce runtime overhead which is inherent to the thread paradigm. In contrast, *protothreads* [3] combine the benefits of both paradigms by providing thread semantics to the programmer while using events at runtime. This is achieved by an automatic code generation step performed by the C preprocessor. However, while using the C preprocessor guarantees portability across C compilers, it also introduces some limitations. For instance, certain C language constructs such as `switch` statements may not be used and values of local variables are not retained across context switches [3]. Furthermore, thread functions are not reentrant, blocking calls may only occur in the top-level thread functions, and debugging is performed in the generated code.

To overcome these limitations, we propose to extend the protothreads abstraction by providing cooperative threads with blocking I/O, reentrant functions, and arbitrary nesting of function calls. This is achieved by a comprehensive compiler which translates thread-based code into efficient event-based code. In order to guarantee the efficiency of the generated code there are still some limitations, though. First, the exact number of threads must be known at compile time. Second, recursive functions must not invoke blocking functions. And third, function pointers must not be used to invoke functions that directly or indirectly invoke blocking functions. We argue, though, that these limitations do not severely affect programming of motes as these constructs are rarely used. Furthermore, the compiler can reliably detect any violations of those restrictions.

2 Our contributions

The main contributions of this work are a platform-independent compiler, which translates thread-based C code (TC) into efficient C code for any event-based runtime environment (EC), a proof of correctness of the transformation, measurements of efficiency which can be used to compare the generated code with hand-written code, and techniques to enable debugging of TC code.

The main challenges related to the transformation of TC into EC code stem from the fact that a call to a blocking function such as `read` or `sleep` must be rewritten into triggering the operation and registering the continuation, which is executed by the runtime environment upon completion of a blocking operation. Every function which potentially calls a blocking function, either directly or indirectly, is affected by the transformation. We denominate such functions *critical functions* and a *critical call* is a call of a critical function.

The transformation of a critical call into a corresponding split-phase operation has two major implications. First, automatic variables in critical functions that are set before the critical call and read afterwards must be preserved. Second, the code following a critical call must be callable. Preserving the state of automatic variables can be achieved by conflating them in a function-specific state structure and by generating appropriate code to access them. If, according to static code analysis, the function is potentially executed by multiple threads at the same time, then there must exist one instance of the structure for each involved thread. Furthermore, a mapping from the current thread ID to the index of the corresponding state structure must be performed. This approach requires to statically allocate the memory for the structure at compile time, even if at runtime a function might actually never be executed by multiple threads. However, since at compile time it is undecidable whether concurrent execution will occur at runtime, the memory for the worst case must be available anyway.

The code surrounding a critical call is split into two separate functions, the second being a callable continuation. The presence of control flow statements induces further splits. In general, additional split points are dictated by jumps and labels of the corresponding GOTO program while jumps are replaced with function calls. The return code of a critical function must know which function it must call in order to continue execution properly. This information is simply added to the function-specific state structure and is set by the caller. In the end, whenever a blocking call occurs, the stack contents of the imaginary TC program is stored in the state structures of the EC code. And as recursion of critical functions is forbidden this is always possible. Figure 1 shows an example of the above-described transformation of TC code (left) into EC code (right). Note, that there is no TC runtime and thus no TC implementations of blocking functions. Instead, there must be proper EC implementations available, which depend on the underlying runtime environment and form the platform abstraction layer of the compiler.

The generated EC code does not induce any overhead for context switching and dynamic memory allocation and is therefore as efficient as hand-written event-based code. Furthermore, it requires only one stack, which is used by the event dispatcher thread and the amount of allocated static memory is the same that would be needed by equivalent hand-written code. Finally, in many cases optimizations are possible. For example, if there is no reentrance there is no need to map from thread ID to state array index and to lookup for the continuation because both results can be determined statically.

By enabling TC-based programming, we allow wireless sensor networks application developers to overcome one of the major drawbacks of event-based programming: the need for explicitly dealing with the scattering of the flow of control over multiple event handlers, the preservation of states across context switches, and the maintenance of continuation information. Our TC compiler provides for automatically generated,

efficient EC code that does not induce additional runtime costs. Furthermore, our compiler can support debuggers operating at the TC level by including proper debugging information in the generated code. In summary, TC programmers can easily develop applications without considering the event-based nature of the underlying system and without worrying about efficiency.

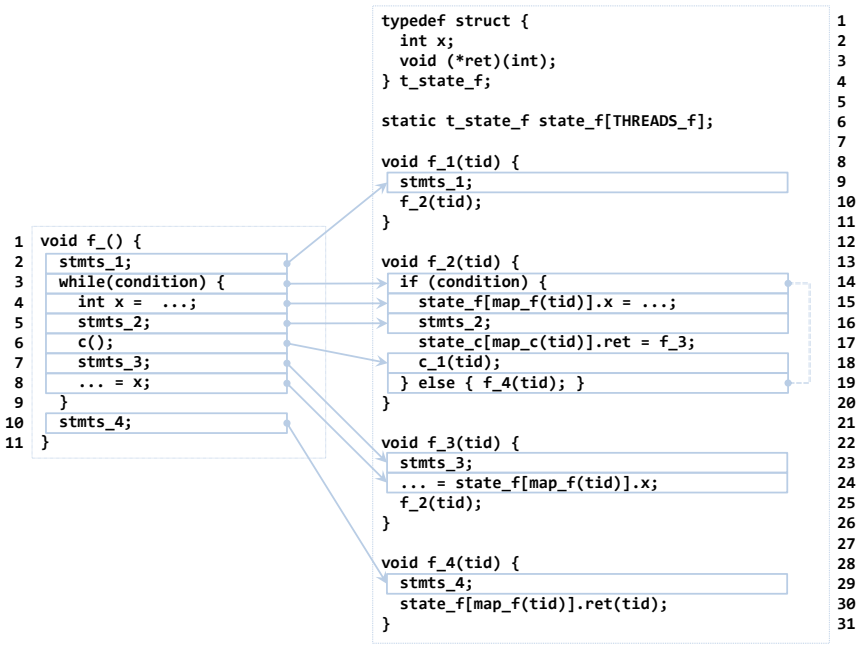

Fig. 1. TC to EC transformation. `stmts` stands for an arbitrary number of statements, `c` is a critical function, `tid` is the current thread ID, `THREADS_f` is the number of threads which use `f` in common, `map_f` and `map_c` map from thread ID to array index. `f` and `c` are assumed to be used more than once in the TC program.

References

1. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for networked sensors. SIGPLAN Not. (2000)
2. Dunkels, A., Gronvall, B., Voigt, T.: Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In: LCN 2004, IEEE Computer Society
3. Dunkels, A., Schmidt, O., Voigt, T., Ali, M.: Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems. In: SenSys 2006
4. McCartney, W., Sridhar, N.: Abstractions for Safe Concurrent Programming in Networked Embedded Systems. In: SenSys 2006
5. Duffy, C., Roedig, U., Herbert, J., Sreenan, C.J.: Adding Preemption to TinyOS. In: EmNets 2007
6. Klues, K., Liang, C.J.M., Paek, J., Musaloiu-E, R., Levis, P., Terzis, A., Govindan, R.: TOSThreads: Thread-Safe and Non-Invasive Preemption in TinyOS. In: SenSys 2009