

Threads2Events: An Automatic Code Generation Approach

Alexander Bernauer*

bernauer@inf.ethz.ch

Kay Römer*†

roemer@inf.ethz.ch

Silvia Santini*

santinis@inf.ethz.ch

Junyan Ma*

junyanma@inf.ethz.ch

*Institute for Pervasive Computing, ETH Zurich, Zurich, Switzerland

†Institute of Computer Engineering, University of Lübeck, Germany

Abstract

There is a long-standing dispute on whether and when thread-based programming should be preferred over the event-based paradigm. This dispute has also extended into the wireless sensor networks domain. Many existing operating systems rely on events due to their efficiency, but make code management difficult. Others rely on threads for developer comfort, but at the cost of reduced runtime efficiency. In this paper we try to combine the best of both worlds by offering a full-fledged cooperative thread abstraction with blocking I/O to the C programmer that is compiled into efficient event-based code. We present the basic code transformations and investigate their efficiency using a representative application case study. We find that RAM usage of generated code competes with hand-written code, but further optimizations are required to reduce the code size and the number of CPU cycles.

Categories and Subject Descriptors

D.1.4 [Programming Techniques]: Sequential Programming; D.3.3 [Programming Languages]: Language Constructs and Features—*Concurrent programming structures*; F.3.2 [Logics and Meanings of Programs]: Semantics of Programming Languages—*Operational semantics*

General Terms

Languages, Measurement, Theory

Keywords

Threads, Events, Compiler, Programming-Abstraction

1 Introduction

Event-driven programming has traditionally been the paradigm of choice to program resource-constrained sensor nodes [5, 7]. The use of events indeed allows to efficiently handle the concurrency-intensive operations such devices must typically deal with [7]. In contrast, using threads

in such a high-concurrency context induces a significant overhead in terms of memory usage and number of context switches. Therefore, although some thread-based solutions exist [2, 3], major operating systems for sensor nodes such as TinyOS or Contiki [5, 7] rely on an event-driven paradigm.

However, event-based programs are notably harder to write, maintain, and debug than their thread-based counterparts [1]. Some authors thus introduced ad-hoc thread libraries to event-based operating systems for sensor nodes in order to facilitate application development [4, 8, 9, 10]. As an alternative, thread-based programs can be automatically translated into equivalent event-based code using an appropriate compiler. Such compiler-based approaches can apply static code optimizations thus producing potentially more efficient code than library-based solutions. Existing approaches such as [6] and [11] have a number of limitations, though. To the best of our knowledge, our system, which we discuss below, is the only to compile almost arbitrary cooperative-thread-based code written in the C language to event-based code for any event-based runtime environment.

Since the thread semantics are implemented at compiler level, there are the following limitations. First, the 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 sensor nodes. 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 sensor nodes, 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 (i.e., if case one, call function A; if case two, call function B; ...) with moderate overhead. What's most important about these constraints is the fact that the compiler can reliably detect violations of them.

The remainder of this paper is organized as follows. After briefly reviewing related work in section 2, we detail how we can automate the threads-to-events code transformation step in section 3. The preliminary evaluation of our system, presented in section 4, shows among other things that the generated code can compete with hand-written code in terms of RAM consumption. Finally, we provide our conclusions

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

HotEmNets 2010, June 28–29, 2010, Killarney, Ireland.

Copyright 2010 ACM 978-1-4503-0265-4 ...\$10.00

This work has been partially supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322, and by CONET, the Cooperating Objects Network of Excellence, funded by the European Commission under FP7 with contract number FP7-2007-2-224053.

in section 5.

2 Related Work

In the context of wireless sensor networks (WSNs), thread programming on top of event-based systems had first been enabled by the development of ad-hoc thread libraries. For instance, TinyThread [9] provides multi-threading to TinyOS programmers. This library includes a tool to estimate the per-thread stack depth and thus allows to minimize RAM overhead. Similar solutions [4, 8, 10] also provide cooperative or preemptive thread abstractions on top of TinyOS or other mote operating systems. However, since libraries implement the thread semantics at runtime, their usage does not allow to apply static code optimizations at compile-time. In contrast, compiler-based approaches can exploit this potential for optimization.

The probably best-known compiler-based solution in this context is the Protothreads approach by Dunkels et al. [6]. Programs written using Protothreads are transformed into event-based code using 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 [6]. 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.

In contrast, we argue for a comprehensive threads-to-events compiler, as introduced by TinyVT [11]. This approach extends the event-based nesC programming language by introducing a `wait` operator that suspends program execution until a certain event occurs. The `wait` operator also specifies the event handler code that should be executed when the event occurs. In addition, TinyVT provides automatic variables that can be accessed from the event handler code. In contrast to our work, however, TinyVT does not completely hide the event semantics of the underlying system. Furthermore, it was specifically designed for nesC/TinyOS, while our approach supports a wide range of event-based runtime environments. Finally, TinyVT does neither support reentrant calls of functions containing `wait` operators nor event handlers containing `wait` operators.

3 From Threads to Events

The main challenges related to the transformation of thread-based code (T-code) into event-based code (E-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, the code following a critical call must be callable, thus the control flow must be modified appropriately. Second, automatic

variables in critical functions that are set before the critical call and read afterwards must be preserved.

3.1 Control Flow

To be amenable for event-based execution, critical functions have to be split into a part that triggers the critical call, and a part that is invoked upon the completion of the critical call. However, if the critical call is nested in control structures such as loops, the transformation may be more complex and is performed as follows. First, the T-code is transformed into an equivalent GOTO program which is a sequence of commands with labels. It is well known that this transformation is always applicable and actually this is what C compilers do in general. Second, the resulting code is split into basic blocks at all labels and critical calls. Third, each resulting basic block is put into its own E-code sub-function. Then, all GOTO commands and transitions between basic blocks are replaced with calls to the corresponding E-code sub-functions. Furthermore, the call to the critical function is replaced by the call to the corresponding first E-code sub-function. Finally, C code is generated for all those functions.

Consider for example the following function `f` in which `statements_x` represents an arbitrary sequence of C language statements and `cond` is a Boolean expression. The E-code obtained by transforming the GOTO program back into C functions is also shown below.

T-code	GOTO program
<pre>void f() { statements_1; while(cond) { statements_2; crit(); statements_3; } statements_4; }</pre>	<pre>statements_1 loop:IF !cond GOTO end statements_2 CALL crit statements_3 GOTO loop end: statements_4</pre>
E-code	
<pre>void f_1() { statements_1; f_2(); }</pre> <pre>void f_2() { if (!cond) f_4(); else { statements_2; crit_1(); } }</pre>	<pre>void f_3() { statements_3; f_2(); }</pre> <pre>void f_4() { statements_4; }</pre>

Likewise, `crit()` is compiled into a set of C functions, such that upon completion `f_3` is invoked as described in Sect. 3.3. Furthermore, the value of an automatic variable of `f` must be retained across calls of the generated functions `f_x`. This is described in the following section.

3.2 Local Variables

For every critical function a statically allocated *state* structure is generated. This structure holds all automatic variables and all parameters that need to be retained across critical calls. Due to the possibility of reentrant function calls, there is one copy of this state structure for each thread.

The resulting E-code then accesses those state structures instead of automatic variables. Note that non-critical functions are not affected by this transformation.

To minimize the RAM consumption, we allocate the state structures for certain functions to the same memory location. In particular, if a critical function sequentially calls two other critical functions, so that the local variables of the latter two functions cannot be active simultaneously, we can allocate the state structures of those two functions to the same memory location using a union. In contrast, the local variables of a function and its calling function are active simultaneously and thus cannot share the same memory location. Using this principle, we can translate the call graph of a critical function directly into a structurally equivalent nested data structure of C structs (for nested calls) and unions (for sequential calls), holding function state structures as in the following example. Here, `state_x` denotes the state structure of a function `x`. `run_send` is the start method of a thread and `sleep`, `read`, and `send` are blocking functions.

thread	corresponding states
<pre>void run_send() { while(1) { sleep(); aggr(); send(); } } void aggr() { read(); //... }</pre>	<pre>struct { state_run run; union { state_sleep sleep; struct { state_aggr aggr; state_read read; } struct1; state_send send; } union1; } state_send;</pre>

There is one such nested data structure for each thread, representing the call graph starting at the main function of that thread. As multiple threads can make reentrant calls to the same function, there must be a mean to retrieve the structure corresponding to the current thread. To this end, we generate a mapping function that returns a pointer to the proper state structure instance. This mapping is based upon the value of a global variable `tid`, which holds the identifier of the current thread. For instance, for function `aggr` in the example above the mapping function would look like:

```
state_aggr* map_aggr() {
  switch(tid) {
    case SEND_THREAD:
      return &state_send.union1.struct1.aggr;
      //...
  }
}
```

We emphasize again that the above transformation only applies to critical functions. Non-critical functions are just copied from T-code to E-code. In addition, only variables whose values need to be retained across calls of sub-functions are affected by the transformation. These are all variables that are written in one sub-function and read in a subsequent one. All other variables such as iteration variables of `for` loops not containing critical calls remain automatic stack-based variables.

3.3 Returning

If a T-code function is only invoked from one place in the code, then the control flow can be hard-coded into the generated E-code. In contrast, if a function may be called from multiple code locations, an explicit return address needs to be saved. In our compilation framework, we include the return address in the state structure of the invoked function much like a local variable while the same optimizations as described in Sect. 3.2 apply. The caller has to set the return address before calling the function as in the following code referring to the example from Sect. 3.1.

<pre>void f_2() { if (! cond) f_4(); else { statements_2; map_crit()->cont = f_3; crit_1(); } }</pre>	<pre>void f_4() { statements_4; (*map_f()->cont)(); }</pre>
--	--

Likewise, return values of critical functions are also stored in its state structure. After return, the caller can retrieve it from there. So a complete state structure for a critical function `int f(int x)` would look like this:

```
struct state_f {
  void (*cont)(); // continuation
  int res; // result
  int x; // parameter
  ... // other local variables
}
```

3.4 Platform Abstraction Layer

The so-called *platform abstraction layer (PAL)* contains event-based implementations of blocking calls in the T-code. The compiler generates E-code skeletons for these functions, but the actual calls to the underlying event-based runtime environment have to be inserted manually. Besides adapting the API of the runtime environment to the generated E-code, the PAL also enables portability.

As an example consider the blocking function `sleep` in the T-code. There is a hand-written E-code implementation of `sleep_1` and `sleep_2` in the PAL. Furthermore, the compiler generates the state structure `state_sleep` according to the T-code signature of `sleep`, and the map function `map_sleep` according to the call graphs of the program. `sleep_1` uses the event-based API of the runtime environment to set up a timer and register `sleep_2` to be called when the timer fires. `sleep_2` will then call the continuation which was saved in the proper instance of `state_sleep` by the caller of `sleep_1`.

While a thread is waiting for the completion of a blocking function, events related to other threads can of course be dispatched. This requires that a (lightweight) context switch is performed by saving the value of the global variable `tid` holding the current thread ID in `sleep_1` and restoring it in `sleep_2`.

3.5 Interfacing the Compiler

In order to perform the transformation, our compiler needs some additional information besides the T-code. In particular, it must know the set of blocking functions, the total number of threads, and the start functions for each of them.

In the context of this work, we assume the declarations for the blocking calls to be available in a T-code header file. The number of threads and their start functions are specified in the `main` function of the T-code program. More specifically, each invocation of the built-in function `RUN_THREAD` causes the global variable `tid` to be incremented. The pointer to the start function of the thread is passed as an argument to the `RUN_THREAD` function. This start function will eventually call a blocking function and return. Once all threads have been started in this way, the generated E-code variant of `main` passes control to the event dispatcher loop of the event-based runtime environment.

4 Evaluation

To verify the feasibility and the practicability of the proposed code transformation, we performed a preliminary experimental study using an application that performs data storage and collection. We started with a *hand-written* event-based implementation of that application. We then rewrote the same application using cooperative threads with blocking I/O. The purpose of this *thread-based* implementation was to serve as input to a manually performed code transformation as described in section 3, resulting in a *generated* event-based version of the application. In the following, we compare the efficiency of the generated event-based code with respect to the hand-written E-code.

The considered test application performs distributed data aggregation with local storage using three main tasks. First, it continuously reads a sensor and stores the values in flash memory (*collect* task). Second, it periodically receives radio messages from other nodes containing their sensor readings and also stores them on flash memory (*receive* task). Finally, it regularly reads the flash memory to compute minimum and maximum readings and sends the results over the radio (*send* task). In the thread-based application, each of these three tasks is performed by a separate thread. Furthermore, the function that writes data to the flash is reentrant since it is invoked by both the receive and the collect threads.

In order to evaluate the correctness of the transformation we first executed both implementations of our test application in our own simulation environment. By analyzing the corresponding API call traces we were able to verify that they behave equivalently. To this end, we choose TinyOS 2.x as a model for the underlying runtime environment. However, since the transformation applies solely to application code, only the API of the operating system but not its implementation is of significance. Therefore, we simply manually compiled the necessary TinyOS interfaces (e.g., `Read`, `Send`) to C code along the lines of the nesC compiler.

Compared to directly working with TinyOS, this approach gave us greater flexibility as we could provide different implementations of the operating system, depending on the needs of the various tests and measurements we per-

buffer size	hand-written	generated	ratio
30	367	429	1.17
50	427	469	1.10
100	577	569	0.99

Table 1. RAM consumption (bytes) of hand-written and generated code.

formed. The complete source code of the study along with more detailed documentation is publicly available¹.

Besides verifying the feasibility of our approach we also compared hand-written and generated code in terms of consumption of ROM, RAM, and CPU resources on a sensor node equipped with an AVR ATmega128.

4.1 Results

We cross-compiled the hand-written and the generated event-based application code using `avr-gcc` version 4.1.2 for the `atmega128` microcontroller with optimization option `-O2`. We then linked the cross-compiled code against an empty implementation of the operating system. The size of the DATA and the BSS sections of the resulting binaries then allows to directly compare the RAM consumption of the two applications. Table 1 shows the resulting sums of both sections and the ratios for different sizes of I/O buffers (see Sect. 4.2 for a discussion of I/O buffers). Note that the numbers for the generated application include the PAL (see Sect. 3.4).

The size of the TEXT sections of the cross-compiled object files indicate how much flash is needed to store the program in the microcontroller. The results are 3166 bytes for the generated application vs. 1080 bytes for the hand-written one, which is a ratio of 2.93. As the PAL is extra code which does not exist in the hand-written case it influences the estimation of the code transformation. Excluding the PAL, the code size of the generated code reduces to 1684 bytes or a ratio of 1.56.

We finally measured the consumption of CPU resources by running the applications in the `Avrora`² simulator. For that purpose we wrote a simple event dispatcher implementation of the operating system. Additionally, we introduced a new main function that after setting up the environment makes each thread perform one iteration of its periodic work.

The results are 4875 CPU cycles for the generated code vs. 1679 cycles for the hand-written code, making for a ratio of 2.78. In order to measure the properties of the code transformation itself we confined the sums on cycles which are executed in the application code, excluding the setup phase and the PAL. Then we measured 1368 vs. 633 cycles, making for a ratio of 2.16.

4.2 Discussion

The selected application represents a “stress-test” for our compilation framework as the code mostly consists of invocations of blocking calls without much “computations”. As in our framework overhead results from blocking calls but not from computations, this application can be considered a worst-case scenario.

¹http://github.com/copton/tc-case_study

²<http://compilers.cs.ucla.edu/avrora/>

Another important issue regarding the interpretation of the performance results is the implementation of the PAL. As discussed in Sect. 3.4, the PAL needs to perform a context switch by saving and restoring the current thread ID. Unfortunately, the TinyOS API does not allow to pass parameters to callback functions when triggering the split-phase operation. Such a mechanism would allow a very efficient implementation of context switches. Instead, our PAL implements a rather inefficient mapping from callback handles to thread identifiers.

For RAM consumption, though, the results for the generated code including the PAL are comparable to those for the hand-written code. This is due to memory sharing of state structures holding local variables in the E-code. The measurements for different sizes of the I/O buffers make this apparent. In the hand-written application, each buffer is a separate static variable in a translation unit. In the thread-based application, these buffers are automatic variables and thus become part of the state structures where they share memory. Thus with larger I/O buffers the ratio becomes smaller to a point where the generated code even becomes more efficient than the hand-written one.

The code size overhead can be explained by the very nature of the code transformation, as every critical function results in the generation of several sub-functions and a map function. The bottom line is that the number of functions is roughly doubled. As the application does not do much computation, the administrative overhead of each single function is significant. Thus doubling the number of functions results in roughly twice the code size.

As the ratio of the cycle counts is higher than the ratio of the code size, the generated code must consist of more expensive instructions on average. According to the AVR ATmega128 datasheet³, this is due to the `call` and `ret` instructions which have a very high code-size-to-cycles ratio. This affects the generated code because it contains, as we outlined above, many function calls.

We expect realistic applications with more computations to result in better ratios for code size and cycles. However, we also expect the ratios to increase if the number of critical calls per function increases. A more thorough study of those factors is subject of ongoing work.

In general, we need to investigate further optimizations to reduce the number of generated E-code functions. For example, the transformations of the application used in the evaluation resulted in sub-functions which do nothing but call another sub-function. Clearly, this could be avoided. Along these lines we believe that in several cases the transformation could result in more efficient code than in the general case as described in section 3.

As pointed out in [3], in the context of WSNs it is often acceptable to trade off lower RAM footprint against a higher execution time. In our case, efficient storage of the state structures has to be paid with the costs of the map functions.

5 Conclusions

We provided the design of a comprehensive compilation framework that automatically generates event-based code from thread-based programs. We tested the feasibility of our approach on a simple but representative case study that can be considered a worst-case scenario. Our preliminary experimental results show that the generated code can compete with hand-written code in terms of RAM usage, while both code size and execution time are larger. We believe, though, that additional optimizations are possible and will reduce both ratios. Furthermore, we expect the ratios to decrease for applications performing longer and more complex computations.

6 References

- [1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *Proc. of the 2002 USENIX Annual Technical Conference*, pages 289–302, June 2002.
- [2] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. Mantis OS: An embedded multithreaded operating system for wireless micro sensor platforms. *Mobile Networks & Applications*, 10:563–579, August 2005.
- [3] N. Brouwers, P. Corke, and K. Langendoen. Darjeeling: a java compatible virtual machine for microcontrollers. In *Proc. of the ACM/IFIP/USENIX Middleware 2008 Conference Companion (Companion 2008)*, pages 18–23, December 2008.
- [4] C. Duffy, U. Roedig, J. Herbert, and C. J. Sreenan. Adding Preemption to TinyOS. In *Proceedings of the 4th workshop on Embedded networked sensors (EmNets 2007)*, pages 88–92, December 2007.
- [5] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proc. of the 29th Annual IEEE Intl. Conf. on Local Computer Networks (LCN'04)*, pages 455–462, November 2004.
- [6] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems. In *Proc. of the 4th Intl. Conf. on Embedded Networked Sensor Systems (SenSys'06)*, pages 29–42, October 2006.
- [7] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. *SIGPLAN Notices*, 35(11):93–104, November 2000.
- [8] K. Klues, C.-J. M. Liang, J. Paek, R. Musäloiu-E, P. Levis, A. Terzis, and R. Govindan. TOSThreads: Thread-Safe and Non-Invasive Preemption in TinyOS. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys'09)*, pages 127–140, New York, NY, USA, 2009. ACM.
- [9] W. P. McCartney and N. Sridhar. Abstractions for Safe Concurrent Programming in Networked Embedded Systems. In *Proc. of the 4th Intl. Conf. on Embedded Networked Sensor Systems (SenSys'06)*, pages 167–180, October 2006.
- [10] C. Nitta, R. Pandey, and Y. Ramin. Y-threads: Supporting concurrency in wireless sensor networks. In *Proc. of the Intl. Conf. on Distributed Computing in Sensor Systems (DCOSS'06)*, pages 169–184, June 2006.
- [11] J. Sallai, M. Maróti, and A. 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, Berlin, Heidelberg, 2007. Springer-Verlag.

³<http://www.atmel.com/atmel/acrobat/doc2467.pdf>