A Comprehensive Compiler-Assisted Thread Abstraction for Resource-Constrained Systems

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

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

ABSTRACT

While size and complexity of sensor networks software has increased significantly in recent years, the hardware capabilities of sensor nodes have been remaining very constrained. The predominant event-based programming paradigm addresses these hardware constraints, but does not scale well with the growing software complexity, often leading to software that is hard-to-manage and error-prone. Thread abstractions could remedy this situation, but existing solutions in sensor networks either provide incomplete thread semantics or introduce a significant resource overhead. This reflects the common understanding that one has to trade expressiveness for efficiency and vice versa. Our work, however, shows that this trade-off is not inherent to resource-constrained systems. We propose a comprehensive compiler-assisted cooperative threading abstraction, where full-fledged thread-based C code is translated to efficient event-based C code that runs atop an event-based operating system such as Contiki or TinyOS. Our evaluation shows that our approach outperforms thread libraries and generates code that is almost as efficient as hand-written event-based code with overheads of 1 % RAM, 2 % CPU, and 3 % ROM.

Categories and Subject Descriptors

D.1.4 [**Programming Techniques**]: Sequential Programming; D.3.4 [**Programming Languages**]: Processors—*Code generation*; D.4.1 [**Operating Systems**]: Process Management—*Threads*

Keywords

Wireless Sensor Networks, Threads, Compiler

1. INTRODUCTION

One of the main challenges in wireless sensor networks (WSNs) is to cope with the scarcity of physical resources. While in other domains Moore's law has led to hardware with ever increasing capabilities, in WSNs advances in technology are applied towards reduced size and cost [21]. Also, energy efficiency is crucial for a long system life-time, which is why today's deployments still operate with 8- or 16- bit MCUs, as a modern 32-bit MCU such as the ARM Cortex-M3 "incurs a $\sim 2\times$ overhead in power draw" [11].

Due to these resource constraints, the event-based programming paradigm is predominant in sensor networks, as it allows implementations of concurrent applications with little memory and computational overhead. While this approach is appropriate for simple sleep-sample-send applications, WSN software is recently gaining complexity with motes running IP stacks [9], HTTP/CoAP services [12], middleware [4], and business logic [18]. Such software includes complex control and data flows that do not fit well with the event-based paradigm. In fact, one has to do both, implement complex control flows via long chains of events and handlers using asynchronous functions, and manually manage data flowing across different event handlers. This often leads to confusing, hard to manage, and error-prone code (e.g., [1, 6, 10, 16, 19, 21]).

For this reason, there have been quite some efforts to enable thread-based programming on sensor nodes, as synchronous functions and sequential computation often lead to simpler and better manageable code. However, these thread libraries are either limited in their expressiveness (e.g. [25]), or they add high overheads compared to event-based programs (e.g. [16]). Compiler-assisted thread abstractions (e.g., [6, 21]) escape this trade-off by translating thread-based programs into equivalent event-based ones, thus exploiting the duality of threads and events [13]. Overall, the application is not only executed by an efficient event-based runtime system but the compiler can additionally apply application-specific optimizations, which is why this approach is promising. Unfortunately, existing compiler-assisted thread abstractions only provide incomplete thread semantics. *Protothreads*, the most popular amongst them, additionally fails to identify invalid input and generates faulty programs instead [6].

This paper presents a comprehensive compiler-assisted thread abstraction that offers a full-fledged cooperative threading model. We thereby target advanced software in the service and application layers where cooperative threads provide a good programming model, as timing issues are of lesser concern. *Ocram*, our compiler implementation, rejects invalid input and otherwise generates event-based code for any event-based runtime environment, requiring only a thin platform abstraction layer to be implemented once. The supported thread model is only marginally constricted due to limits that are inherent to the compiler-based approach.

The contributions of this paper are:

1. A platform-agnostic source-to-source transformation scheme which translates ISO/IEC 9899:1999 (C99) programs using cooperative threads and synchronous functions into equivalent C99 program using events and asynchronous functions,
2. Ocram, a GPL-licensed compiler prototype that implements the transformation,

IPSN'13, April 8–11, 2013, Philadelphia, Pennsylvania, USA.

3. platform abstraction layers to bind Ocram to Contiki and TinyOS,

4. an evaluation which a) shows the feasibility of compiler-assisted threads for three different WSN application archetypes, b) verifies the correctness of the transformation, and c) measures the resource costs of this abstraction compared to both native event-based implementations and thread libraries.

The remainder of the paper is organized as follows: Section 2 and 3 provide the context and an overview of our approach. Section 4 describes the translation scheme while Section 5 covers the interoperability between generated and existing event-based code. Section 6 documents the setup of the conducted experiments and Section 7 presents the results of our evaluation. Finally, Section 8 gives an outlook on future research directions.

2. STATE OF THE ART

Operating systems like *TinyOS* [8] and *Contiki* [5] target small WSN devices by utilizing resource-efficient event dispatching. However, a continuous demand for thread abstractions has lead to different approaches to enable threads on resource-constrained systems. Unfortunately, all of them either have high resource-demands or provide only incomplete or non-standard thread semantics.

Fibers, for instance, allow for exactly one additional thread [25], while *Y-Threads* indeed support multiple threads, but require to manually distinguish "run-to-completion threads" [19]. To implement a comprehensive thread abstractions, the most common way is to have one stack per thread and switch context either in a cooperative [16] or a preemptive fashion [2, 10]. Multiple, preallocated stacks are, however, not resource efficient for two reasons. First, each stack has to provide enough cut-off to support the worst case of occurring interrupt handlers operating on the respective current stack. And second, the size of all preallocated stacks is usually higher than the actual maximum stack consumption of an application because usually not all threads reach their maximum stack usage at the same time. As a consequence, multiple, preallocated stacks do not scale well with the number of threads.

Outside of the WSN domain, von Behren et al. have addressed this problem with *Cappriccio* [24]. This system automatically augments thread-based applications with code for dynamic stack allocation, thus avoiding preallocated stacks. Although this enables high performance and scalability for Internet services on Linux, it is not a viable option for WSNs. First of all, with dynamic stack management it is hard to guarantee that out-of-memory situations do no occur, which is certainly crucial for the reliability of an embedded system. Also, dynamic memory management not only introduces a runtime overhead, but also poses the problem of memory fragmentation, as typical WSN devices are not equipped with a memory management unit (MMU).

In general, threads and events are known to be dual to each other [13]. Adya et al. even mix thread-based and event-based code in a single program by using small adaptors and a special scheduler [1]. As this is a multi-stacked, runtime-based solution, however, it is not resource-efficient. Overall, run-time based thread abstractions entail a trade-off between completeness of the thread semantics and resource overhead compared to event-based programming.

In contrast, compiler-assisted thread abstractions exploit the duality of threads and events by translating a thread-based program into an equivalent event-based program, thus escaping this trade-off. While the developer has the comfort of threads, the resulting program is both single-stacked and avoids the extra context switching overhead of a thread scheduler, as the existing event dispatcher is used instead. Additionally, while a runtime-based solution always has to support the most generic case, static analysis can exploit application-specific optimizations.

The first system that provided a compiler-assisted thread-abstraction for WSNs was Contiki's *protothreads* [6]. Technically, protothreads are a set of C preprocessor macros that enable the syntactical illusion of threads and synchronous functions. As the context of each thread (a.k.a. protothread process) comprises only two bytes, the authors claim to provide the most efficient thread implementation. At a closer look, however, this convincing performance comes at the price of incomplete thread semantics. First of all, the state of automatic variables is not preserved across yield points. Second, yield points can only be placed in the thread start function, which severely restricts the software architecture of protothreads-based applications. In addition, certain user mistakes are not detected at compile time, but have to be indirectly examined by observing unexpected runtime behavior. Such mistakes can be as subtle as using `switch` statements that interfere with those that are expanded from the protothreads macros. As the C preprocessor can only perform local substitution of language tokens, these limitations are inherent to the protothreads abstraction.

TinyVT [21] employs a dedicated compiler. This abstraction has been specifically designed for TinyOS, which provides a component-based software architecture with event-based interfaces. TinyVT enables software developers to implement single components by writing sequential code, as it is possible to inline event handlers in code blocks following a special `await` statement. Also, the runtime system preserves the state of automatic variables. Although TinyVT overcomes many of protothreads' drawbacks, the supported thread semantics is still restricted and deviates substantially from common threading models. 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 cannot be shared between multiple threads and functions cannot be re-entrant.

The most recent compiler-assisted thread abstraction is *UnStacked C* [17], "a source-to-source transformation that can translate multi-threaded programs into stackless continuations." This is a hybrid approach, as the input application uses a thread library such as TOSThreads [10] and the generated application relies on a modified version of this library. The translation reduces the memory requirements of the application in particular by softening the timing guarantees of context switching (so-called *lazy preemption*). Overall, the generated runtime system still depends on a thread library and it remains to be seen if lazy preemption is appropriate for WSN applications.

In summary, runtime-based thread abstractions are not resource-efficient and existing compiler-assisted thread abstractions provide only incomplete or non-standard thread semantics. The core contribution of our work lies in overcoming the seemingly inherent trade-off between completeness of thread semantics and resource overheads that is prevalent in the state of the art. Specifically, our approach offers cooperative threads with only minor restrictions while achieving the efficiency of event-based programming. Our dedicated compiler performs a platform-agnostic source-to-source transformation that exploits application-specific optimizations, and the output application only relies on the existence of an event-based operating system (OS).

Up to our knowledge, similar approaches have not been taken in other embedded systems domains so far, presumably because application complexity is low enough to be manageable with events, or because availability of resources is high enough to allow for thread libraries.

3. OVERVIEW

This sections gives an overview of relevant terms and programming paradigms (Section 3.1), introduces our approach on a high level (Section 3.2), and discusses its fundamental limitations (Section 3.3).

3.1 Programming Paradigms

An application usually consists of multiple *tasks*, which are logical groups of computations that pursue a common goal. Tasks effectively perform their work by calling functions of the *application programming interface* (API) of the OS. In case of a *synchronous* function, the associated operation is guaranteed to have completed when the function returns. In contrast, calling an *asynchronous* function only triggers an operation which will complete eventually.

The reportedly most efficient way to execute multiple tasks virtually in parallel is to follow the *event-based paradigm*. In this paradigm, a task is formed by a causal chain of events and event handler functions, which is frequently initiated by recurring events. As the runtime system waits for a handler function to return before processing the next event, a single stack suffices to execute all tasks. One major source of complexity of this paradigm is that managing the control flow of a task is hard, as the code is spread out over multiple event handlers. A second major source of complexity is that the execution contexts of tasks (i.e., the values of variables) have to be manually managed and preserved between subsequent event handler invocations. Additionally, every function that calls an asynchronous function becomes asynchronous itself, which recursively applies to the whole call stack and forces all functions' implementations to be split up (cf. "stack ripping" [1]). In summary, the event-based programming paradigm is efficient, but requires a lot of cumbersome and error-prone work to be done manually.

In contrast, the *thread-based paradigm* overcomes these problems by means of synchronous OS APIs, where the control flow is sequential, and a task's context can be stored in *automatic* (i.e., function-local) variables using a scope-based automatic live-cycle management. A *thread* is one flow of control that drives the execution of a task. It starts with the invocation of a *thread start function* and sequentially executes the statements of this function.

The reason why threads are considered inefficient is that the context of a task is the complete stack of its thread. As it is generally difficult to estimate the maximum size of a stack, and as hardware interrupt handlers always operate on the current stack and can be invoked at any point in time, a common approach is to add enough cutoff to be safe from out-of-stack situations. On sensor nodes, though, the cutoffs of a reasonable amount of threads quickly sum up beyond the total amount of available memory. This is why providing comprehensive thread abstractions for WSNs is not trivial.

3.2 Our Approach

As depicted in Figure 1, in the context of our work, a dedicated compiler called Ocram translates a thread-based application (*T-code*) into an equivalent event-based application (*E-code*). The T-code application is built upon a synchronous T-code API which has been manually derived from an asynchronous OS API. Instead of implementing the T-code API — which is what runtime-based thread abstractions do — the compiler generates a corresponding E-code API which is used by the generated E-code application. The *platform abstraction layer* (PAL) (cf. Section 5) implements the E-code API by means of the OS API.

Given the call graph of a T-code application, every function that has a path leading to a T-code API function is referred to as *interruptable function* and every edge of such a path is referred to as *interruptable call*. The compiler's task is to turn interruptable functions into event handlers (cf. Section 4.2) and to preserve the value of automatic variables across single invocations if needed (cf. Section 4.1). And of course, the E-code should be equivalent to the T-code (cf. Section 4.3) to ensure that it behaves as intended by the T-code programmer.

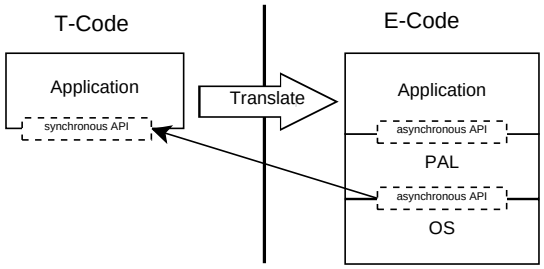


Figure 1: *System Overview*: The compiler translates the thread-based program (T-code) into an equivalent event-based program (E-code). The platform abstraction layer (PAL) mediates between the generated application and the given operating system (OS).

The generated E-code application contains one central event handler function per thread which contains the inlined bodies of all involved interruptable functions. Always after calling an E-code API function, the handler returns control to the PAL which invokes the handler again as soon as the requested event occurs. Resuming the task is possible, as each point of continuation is equipped with a label whose address is taken by the handler, passed to the PAL, and retrieved as a function parameter with the subsequent invocation.

Additionally, the compiler generates a static data structure for each thread which contains all variables that are read after an interruptable call. By replacing automatic variables with their static counterparts, the relevant state of the task stays available. Overall, the transformation of the control flow preserves the order of statement execution while the transformation of the data flow preserves the effect of each statement, which is why T-code and E-code are equivalent.

3.3 Limitations

A major drawback of static analysis and translation is that it is limited to decidable problems. In particular, the compiler has to be able to infer the application's call graph, which implies three things. First, it is not allowed to take the address of an interruptable function. Instead one has to use case differentiation, which causes only moderate overhead. Second, interruptable functions may not be recursive, also because this would turn the estimation of the stack consumption undecidable. For the same reason, however, recursive algorithms are uncommon in embedded systems, which makes this a minor restriction. And third, we can not support dynamic thread creation but have to statically assign threads with thread start functions. As WSN applications tend to be composed of a fixed set of tasks, we don't consider this a severe limitation either.

These limitations are all inherent to compiler-assisted approaches and our compiler reliably rejects invalid T-code. Thus, we think it is appropriate to refer to our approach as a comprehensive thread abstraction.

Besides these limitations, the scarce resources additionally call for cooperative threads as opposed to preemptive ones. In cooperative style, context switching only occurs at yield points, thus the saved context only consists of the variables that are read after the yield point. In contrast, the context in preemptive style consists of the thread's complete stack and all CPU registers.

Of course, cooperative threads can not guarantee timings, pri-

```
double c1(char* s){
  c2(s);
  int i = f();
  c3();
  return i;
}
```

⇒

```
1  typedef struct {
2    void* cont;
3    double result;
4    char* s;
5    int i;
6    union {
7      frame_c2_t c2;
8      frame_c3_t c3;
9    } frames;
10 } frame_c1_t;
```

Figure 2: The *T-stack structure* of an interruptable function. `c1`, `c2`, and `c3` are interruptable functions; `f` is not.

orities, and fairness irrespective of the concrete implementation of each thread. Thus, our approach targets application and service layers of WSN applications where real-time is less of an issue. As, in contrast to general purpose computing devices, motes are usually not deployed with untrusted and thus uncooperative code, requiring cooperative threads is feasible.

Finally, we consider cooperative threads the easier programming model, as shared state has to be checked only after yield points. In contrast, preemptive threads are non-deterministic by default [14], and practice has shown that "humans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code" [23].

4. TRANSLATION SCHEME

The goal of the translation is to transform valid T-code into equivalent E-code. To this end, the translation distinguishes between interruptable and non-interruptable functions. Non-interruptable functions are passed through unchanged and are not considered any further.

Regarding interruptable functions, the first step is the translation into an intermediate representation (IR). Besides some technicalities such as the uniqueness of identifiers this implies two things. First, `while` and `for` loops are replaced by `if` and `goto` statements with labels. And second, interruptable calls only appear in one of two ways. The *first normal form* is a statement consisting of a single interruptable call. In contrast, the *second normal form* contains an arbitrary assignment operator, an l-value without interruptable calls, and a single interruptable call, i.e., anything of the form `expression = interruptable_call(parameters);`.

To establish the normal form, we make use of the fact that interruptable calls in nested expressions can be substituted by new variables that are initialized by the same interruptable call in a directly preceding statement. Special care is needed to handle Boolean short-circuit evaluation correctly, which is achieved by splitting Boolean expressions into a sequence of `if` statements as needed.

4.1 Data Flow

Given the IR, we use Hoopl [20] to perform a liveness analysis on each interruptable function to find the set of *critical variables*, i.e., automatic variables that are possibly read after an interruptable call and thus have to be preserved. Because aliasing turns this into an undecidable problem, the compiler makes conservative choices such as considering a variable as critical if its address is taken somewhere.

The compiler then generates a so-called *T-stack frame* for each interruptable function (cf. Figure 2). Such a frame contains the *continuation*, i.e., information about where execution should continue when the interruptable function returns (line 2), the return value of the function if existent (line 3), its parameters (line 4), its

```
void blinky() {
  while(1) {
      wait();
      // do something
  }
}
```

⇓

```
1  void thread_0(..) {
2  //...
3  start: if (!1) return;
4    stack_blinky.frames.wait.cont = &&label_blinky_1;
5    goto label_wait_0;
6  label_blinky_1:
7    // do something
8    goto start;
9  label_wait_0:
10   // E-code body of wait
11   goto *stack_blinky.frames.wait.cont;
12 }
```

Figure 3: *Interruptable call of an interruptable function*

```
void wait() {
  sleep(23);
  // do something
}
```

⇓

```
1  void thread_0(void* cont) {
2      if (cont) goto *cont;
3  //... E-code body of blinky
4  label_wait_0:
5    stack_blinky.frames.wait.frames.sleep.until = 23;
6    stack_blinky.frames.wait.frames.sleep.cont = \
7        &&label_wait_1;
8    sleep(&stack_blinky.frames.wait.frames.sleep);
9    return;
10 label_wait_1:
11   // do something
12   goto *stack_blinky.frames.wait.cont;
13 }
```

Figure 4: *Interruptable call of a T-code API function*

critical variables (line 5), and a `union` of the T-stack frames of all callees (lines 6–9). Furthermore, one T-stack frame for each thread starting function is instantiated statically and is called the *T-stack* of the thread.

Similarly, the *E-stack frame* of a function contains all of its non-critical variables, and for each thread the `union` of all E-stack frames of the involved functions establish the *E-stack*, which is an automatic variable of the *thread execution function* (cf. next section). Given these stacks, the compiler is rewriting access to variables by replacing them with the corresponding variables from the stacks.

4.2 Control Flow

To translate the control flow, the bodies of all interruptable functions that are used by a thread are inlined into one common *thread execution function* for each thread. This thread execution function serves as a single event handler function for all events concerning the corresponding thread. When inlining, the compiler equips every first statement of a function body and every first statement after an interruptable call with a unique label, which serves as a continuation point. The compiler also translates all interruptable function calls and returns.

Figure 3 depicts an example of how interruptable calls to interruptable functions are replaced by the following sequence of state-

ments: First, the callee's parameters are written to the T-stack (not shown). Second, the continuation information for the callee is written to the T-stack (line 4). Third, a `goto` jumps to the start of the callee's body (line 5). Similarly, every `return` statement in the T-code is replaced by a `goto` statement that uses the continuation information stored on the T-stack (line 11). In case of an interruptable call in second normal form, the function's return value is copied from the T-stack and assigned to the translated l-value of the normal form's assignment (not shown).

Figure 4 shows that interruptable calls to T-code API functions are replaced by a slightly different sequence of statements. First, function parameters (line 5) and continuation information (line 6–7) are also copied to the T-stack. But then, the E-code API function is called, passing it a pointer to its T-stack frame (line 8). Last, the thread execution function returns in order to pass control back to the PAL (line 9).

The PAL takes care to call the thread execution function (called `thread_0` in this example) back as soon as the operation has completed, passing it the continuation that has previously been copied to the T-stack of the API function. The first statement in each thread execution function is a `goto` (line 2) that resumes the computation at this location. Again, the return value of the API function can be copied from its T-stack frame if necessary (not shown).

4.3 Equivalence and Correctness

In order to reason about the correctness of the transformation, we define an event-based application to be *equivalent* to a thread-based application if and only if every possible *observable behavior* of the E-code corresponds to one possible observable behavior of the T-code, assuming we would actually execute the T-code. The intuition behind this definition is twofold. First, if from an observer's point of view, the interactions with the environment performed by the E-code and T-code are indistinguishable from each other, then both variants apparently do "the same thing" and it does not matter which one is actually executed. And second, if every observed interaction of the E-code can be explained by a hypothetical execution of the T-code, then "nothing surprising" can happen.

The *observable behavior* is defined as the order of all API calls including the *values* of all input parameters. Note however, that the exact timing of the API calls is not part of the observable behavior, as cooperative threads are not viable for timing-critical tasks. The *value* of a parameter is defined as the bit representation of a variable of primitive type, recursively applied to structures and unions, and the value of a pointer is the value of the referenced object.

When comparing the observable behavior, we refer to corresponding functions from the T-code API and the E-code API. As the latter is systematically generated from the former, our definition of equivalence is sound.

The transformation of the data flow leaves the observable behavior unchanged because the variables that live on the stacks can simulate the life cycle of the corresponding automatic variables. First, this is because E-stacks contain only non-critical variables and are automatic variables themselves. And second, T-stacks simulate runtime stacks of threads by design. So overall, the translation simply exchanges the memory locations of variables while all statements formed with these variables keep their effects.

Concerning the transformation of the control flow, the transformation preserves both the sequence of statement between two yield points and the continuation of each interruptable call, which maintains the equivalence of each of these steps. And as the execution of T-code is non-deterministic with regard to the order of interleaving tasks, for each possible sequence of events in the E-code there is

```
1    | int dt = 500;
2  A | int get_leds() { /* ... */ }
3    | void set_leds(int state) { /* ... */ }
4    | int time(); // included from OS header file
5
6  H | __attribute__((tc_api)) void sleep(int until);
7
8      __attribute__((tc_thread)) void blinky()
9      {
10   |   unsigned char state;
11   |   state = get_leds();
12   |   while(1) {
13 F |     wait();
14   |     state ^= 0xff;
15   |     set_leds(state);
16   |   }
17     }
18
19     void wait() {
20   |     int now;
21 G |     now = time();
22   |     sleep(now + dt);
23     }
```

⇓

```
1    | int dt = 500;
2  A | int get_leds() { /* ... */ }
3    | void set_leds(int state) { /* ... */ }
4    | int time();
5
6    | typedef struct {
7    |   void* cont; int until;
8    | } frame_sleep_t;
9    | typedef struct {
10   |   void* cont; int now;
11   |   union { frame_sleep_t sleep; } frames;
12 B | } frame_wait_t;
13   | typedef struct {
14   |   unsigned char state;
15   |   union { frame_wait_t wait; } frames;
16   | } frame_blinky_t;
17
18 C | frame_blinky_t tstack_blinky;
19
20 D | typedef struct { int now; } eframe_wait_t;
21
22 H | void sleep(frame_sleep_t*);
23
24   | void thread_0(void* cont) {
25 E |   union { eframe_wait_t wait; } estack;
26
27   |   if (cont) goto *cont;
28
29   |   tstack_blinky.state = get_leds();
30   | start:
31   |   if (!1) return;
32   |   tstack_blinky.frames.wait.cont = &&label_blinky_1;
33   |   goto label_wait_0;
34 F | label_blinky_1:
35   |   tstack_blinky.state ^= 0xff;
36   |   set_leds(tstack_blinky.state);
37   |   goto start;
38
39   | label_wait_0:
40   |   estack.wait.now = time();
41   |   tstack_blinky.frames.wait.frames.sleep.until = \
42   |       estack.wait.now + dt;
43 G |   tstack_blinky.frames.wait.frames.sleep.cont = \
44   |       &&label_wait_1;
45   |   sleep(&tstack_blinky.frames.wait.frames.sleep);
46   |   return;
47   | label_wait_1:
48   |   goto *tstack_blinky.frames.wait.cont;
49     }
```

Figure 5: *A minimal, but complete example:* Translating T-code (top) to E-code (bottom).

```
typedef enum {
  APICALL_sleep,
  /* constants for other API calls */
} APICall;

typedef struct {
  union {
      struct {
          frame_sleep_t* frame;
          struct etimer et;
      } sleep;
      /* contexts of other API calls */
  } ctx;
  APICall apicall;
} ThreadContext;

ThreadContext threads[1];
ThreadContext* cur_thread;

void sleep(frame_sleep_t* fr) {
  cur_thread->ctx.sleep.frame = fr;
  cur_thread->apicall = APICALL_sleep;
  clock_time_t now = clock_time();
  etimer_set(&cur_thread->ctx.sleep.et, fr->until-now);
}
```

Figure 6: *PAL implementation of* `sleep`

one possible control flow in the T-code. From this we can deduce the overall equivalence.

4.4 Example

Figure 5 shows a minimal, but complete example. First, the lines labeled with A in both listings show that everything that is neither an interruptable function nor a T-code API function is passed from T-code to E-code unchanged.

Label B shows the T-stack frame structures for the interruptable functions `sleep`, `wait`, and `blinky`. For example, the integer `state` (E-code, line 14) originates from the automatic variable `state` of the function `blinky` (T-code, line 10). Label C shows the instantiation of the T-stack.

Label D shows the E-stack frame for the function `wait` which is the only function with a non-critical variable. Thus, the E-stack (Label E) contains only one member.

Label F and G show the body of the function `blinky` and `wait`, respectively, and we can see two modifications. First, access to variables and parameters is altered. For instance, access to `state` and `now` (T-code, line 11 and 21) gets translated to access to the T-stack and E-stack (E-code, line 29 and 40). And second, the control flow is translated into continuation passing style. This involves instrumenting the code with labels marking the single continuations (E-code, line 34, 39, and 47). It also involves rewriting interruptable calls to interruptable functions (E-code, line 32 and 33) and to E-code API functions (E-code, line 41–46).

Finally, label H shows how declarations of API functions get translated.

5. PLATFORM ABSTRACTION LAYER

As already mentioned in Section 3.2, the PAL has to implement the E-code API by means of the OS API. But it is important to note that the PAL is not a conceptual requirement of our compiler-assisted approach to cooperative threads. Instead, it is only required to use Ocram with existing operating systems. In general, however, one could very well design an OS that already provides the systematic E-code API as assumed by Ocram. In this case no PAL would be needed at all.

The complexity of the PAL directly depends on how similar the existing OS API is to the systematic E-code API. Likewise, the integration of existing native code with generated E-code is also handled by the PAL in an OS-specific manner.

```
static char event_handler_0(struct pt* ptinfo,
        process_event ev, process_data_t data)
{
  void* cont;
  cur_thread = &threads[0];
  if (ptinfo->lc == 0) { // first invokation
    ptinfo->lc = 1;
    cont = NULL;
  } else if (cur_thread->apicall == APICALL_sleep
          && ev == PROCESS_EVENT_TIMER
          && data == &cur_thread->ctx.sleep.et) {
    cont = cur_thread->ctx.sleep.frame->ec_cont;
  } /* else handle other API calls */
  thread_0(cont);
  return PT_YIELDED;
}
```

Figure 7: *PAL event handler running a T-code thread.*

To illustrate the building blocks of a Contiki PAL that we used for our evaluation (cf. Section 6), we assume a minimalistic T-code example employing only a single thread and the single API function `sleep` from Figure 5. As already depicted there, the translation generates `frame_sleep_t` and the PAL needs to implement the E-code API function `sleep(frame_sleep_t* frame)` and ensure a proper thread continuation.

Despite protothreads, Contiki builds upon an event-based approach with a single event handler function for each task. To clarify this, we avoided using any of the protothreads macros for the Contiki PAL. We are still using what is called a "process" in Contiki, as this is unavoidable, but without protothreads, a process is nothing more than an event handler and some meta information.

Figure 6 shows the implementation of `sleep` and Figure 7 shows the event handler that executes the single T-code thread. We can tell that the event handler has been called for the very first time by looking at the meta information in `ptinfo`. We memorize this case and call the generated thread execution function `thread_0` with a `NULL` continuation, thus starting the task. Soon after that, the task calls `sleep`, which sets a timer and returns. So does the thread execution function and the PAL, leaving the system to wait for the PAL being called again with a timer event. As soon as this happens, the PAL calls the thread execution function with the registered continuation, thus resuming the task. As the PAL uses standard Contiki primitives to communicate with other modules, integrating existing native code poses no problems.

PALs for other operating systems of course look different, but in general it should always be possible to perform the necessary mapping. For example, our proof-of-concept implementation of a TinyOS PAL employs one instance of a special component per T-code thread and each of these components is wired to whatever components it needs to implement the E-code API. API functions like `sleep` are implemented similarly to Figure 6, but the occurrence of an anticipated event results in a task being posted, which then calls the thread execution function.

6. EVALUATION

In order to evaluate our approach we have written Ocram, a Haskell-based implementation of the translation scheme described in Section 4. The source code of Ocram and of the complete evaluation is published[1] under a GPL license.

[1] `https://github.com/copton/ocram`

We have additionally chosen a set of three case studies, each following a real WSN application archetype (cf. Section 6.2), and overall covering a representative range of application types, programming concepts and concurrency patterns. We implemented each of these case studies in three variants (cf. Section 6.1): 1) a native event-based version, 2) a thread-based version using a thread library, and 3) a T-code version using Ocram. All nine resulting programs are written for Contiki and executed via COOJA/MSPSim, while an extra COOJA plugin collects various logs and measurements. Section 6.3 explains how we used the logs to verify the correctness of the applications and the transformation and Section 6.4 describes which measurements we took how.

6.1 Variants

For each case study application we first implemented a *native* (NAT) variant using the event-based paradigm. To this end, we either copied existing code or wrote an implementation following common programing patterns as encountered in the Contiki community. This implies using protothreads, which disguises that the runtime system is event-based and adds an overhead of two bytes per protothread. We argue, though, that this does not significantly bias the ground truth of our evaluation, which is the performance of a native event-based application.

In Section 2 we have argued why compiler-assisted thread abstractions are more efficient than runtime-based solutions. To verify this hypothesis, we have secondly written a threaded variant of the application which is directly executed using a *thread library* (TL). For this purpose, we have ported the TinyThreads [16] thread library to Contiki, as it is the only available full-fledged thread library for cooperative threads in sensor networks. We kept the basic context switching code and the general scheduler architecture, but we had to adapt the details to Contiki's APIs. We also removed support for preemption and dynamic thread creation and termination to avoid extra overhead for features that our abstraction does not provide. Overall, a single protothread executes the scheduler and all application-level threads.

Finally, the third variant is the E-code application *generated* (GEN) from a thread-based T-code application. To enable a fair comparison, the three variants only differ from each other if the different programming models require so. We thus share most non-interruptable code via separate translation units and we also copied as many source code lines between the variants as possible.

6.2 Case Study Applications

Figure 8 shows simplified pseudo code for the three case study applications. The underlined functions are thread start functions with an endless loop executing the listed code, while the italic functions are re-entrant interruptable functions. The three applications use the API functions `sleep`, `receive`, and `wait` and all three of them can be interrupted via `notify`.

The first case study is a typical *data collection and in-network aggregation* (DCA) application consisting of three tasks. Overall, the application reads values from the local sensor, receives values from its child node(s) and sends aggregated values to its parent node. This constitutes a consumer-producer pattern with inter-thread communication via a shared ring buffer, but no explicit thread synchronization and no re-entrant code. The major architecture of this case study can be found in many deployments (e.g. [7]).

The second case study is a complete client-side implementation of the *CoAP protocol* [22] including application-level payload fragmentation via stop-and-wait [3], and a minimal application layer. The program consists of three tasks and overall the client repeatedly sends PUT and GET requests to the server. The PUT request sets the seed for a random resource on the server, while the GET request retrieves a possibly large sequence of characters from this resource. Both the value of the seed and the length of the character sequence are chosen randomly. This application involves both explicit thread synchronization via `wait` and `notify` and re-entrant code. The native implementation for this study is Contiki's CoAP implementation [12].

```
receiving:
 receive
 add values to ring buffer

collecting:
 sleep 23 second
 read from sensor
 add value to ring buffer

sending:
 sleep 127 seconds
 empty ring buffer
 aggregate values
 send results to parent
```

(a) DCA

```
server:
 receive
 if notified:
  send response
 else:
  notify available worker

worker [1-N]:
 wait
 handle call

handle call:
 if read fast sensor:
  read sensor
 if read slow sensor:
  sleep // emulation
  read sensor
 if tell:
  send call to peer
  receive response
 notify server
```

(c) RPC

```
transactions:
 sleep until next timeout
 if notified:
  accept new transaction
 else:
  send transaction
  double transaction timeout

receiver:
 receive
 cancel pending transaction
 notify stop-and-wait

client:
 sleep 10 seconds
 stop-and-wait of PUT
 stop-and-wait of GET

stop-and-wait:
 for each fragment:
  create transaction
  send transaction
  wait
```

(b) CoAP

Figure 8: Pseudo code of the *case study applications*.

The third case study is motivated by a programming framework for sensor networks [18] that offers so-called *tell actions*, where a node may tell one or more other nodes to execute a (potentially synchronous) command and the tell action itself waits until all nodes have finished executing the command. In other words, a tell action basically is a synchronous one-to-many *remote procedure call* (RPC). In a network in which multiple tell actions can occur at any time, each node should be able to handle multiple RPCs at the same time. This calls for a thread pool, a common concurrency pattern that can be found in many RPC systems such as CORBA. Although there are some RPC frameworks for sensor networks [15, 26], none of them supports concurrent invocations. Thus, we implemented this framework from scratch.

In order to keep the focus on the programming model, we only support three basic remote calls that conceptually cover the whole spectrum of interest: 1) reading a value from a fast sensor such as a temperature sensor, 2) reading a value from a slow sensor, which involves some startup time and thus a synchronous function on the callee's side, and 3) a tell operation, which involves delegating any

of these three remote calls to a different node and thus also requires a synchronous function on the callee's side. The application implements both the client and the server side of the RPC protocol and involves explicit thread synchronization and re-entrant functions.

6.3 Verification

The verification serves three purposes. First, we want to make sure that we measure only the effects of the different programming abstractions. Second, we want to test each variant of each application for bugs. And third, we want to verify the correctness of the transformation.

To preserve the fairness of the comparison, we used the same COOJA simulation for all variants, only exchanging the binary under test in each case. This means that spatial mote distribution, radio model, neighbor nodes, random seeds, etc. are constant. That is, the execution environment is deterministic and produces the same results repeatably. Additionally, we compared the source code of the three variants to make sure that they are not needlessly dissimilar.

In order to verify the executed applications, we wrote a COOJA plugin that collects a log of `printf` traces, which serves as an input to an application-specific verification script. And concerning the correctness of the transformation, the plugin also collects a log of the observable behavior (OB) as defined in Section 4.3. The TL variant de facto constitutes the execution of the T-code, while GEN is the execution of the E-code. Thus, comparing the OB of TL and GEN ensures the equality of T-code and E-code and thus the correctness of the transformation.

6.4 Measurements

As a first and simplest measurement, we counted the *lines of code* required to implement the application logic for each variant. Lines of code is in general not very significant, but when using identical code formatting rules, as we did in our evaluation, it provides a quick but good estimation of the expressiveness of the programming model. In order to focus on the effects of the different programming models, we did not include the shared translation units into the counting. Similarly, we neither took GEN's PAL nor TL's scheduler into account, as this code needs to be written only once and can thus be regarded as being part of the operating system, which we did not count either.

Next, we compiled each application for the Tmote Sky platform using the MSP430 port of the GNU Compiler Collection (GCC) version 3.2.3. The compilation process was performed by Contiki's build system with `SMALL=1`, which amongst other things instructs the linker to remove unused functions. To obtain static measurements from the resulting binary we used `objdump` from GCC's binutils and retrieved the size of the *text* section (i.e., the machine code itself), the size of the initialized *data* section, and the size of the uninitialized data section, also known as *bss* section.

Besides these static measurements, we were also interested in runtime properties. To obtain a precise count of *CPU cycles*, we modified the Contiki system by introducing a `volatile void*` variable called `proc_hook`. Right before invoking a process, the scheduler writes the address of the descriptor of the particular process to `proc_hook`. And right after the process returns control, `proc_hook` is set to `NULL`. As `proc_hook` is declared `volatile`, its modifications are guaranteed to happen at the intended moments and in the right order. Our plugin can thus install a break point for updates to `proc_hook`, which enables it to precisely sum up CPU cycles for each process individually.

All interrupt functions and the code that logs the OB signal their invocation via `proc_hook` as well. The plugin can thus remove these CPU cycles from the current process' account, and by considering the cycles for the prologue and the epilogue of the interrupt functions and the cycles for the write operations to `proc_hook`, it measures the exact number of CPU cycles per process.

For NAT, we counted the CPU cycles of all processes that run an application task, leaving out any OS processes. For TL, we counted the CPU cycles of the single scheduler process only, as it executes all application threads. And finally, for GEN, we counted the CPU cycles of all Contiki processes that execute an application thread (cf. Section 5). Overall, the counted CPU cycles cover the same application functionality in each case.

In order to measure the *maximum stack consumption*, our plugin installs a break point for updates to the stack pointer register (SP). For NAT and GEN, tracking the maximum SP value and subtracting it from the start address of the stack is sufficient. For TL, we also need to take the stacks of the application threads into account, though. Our plugin does this accurately and thus obtains the precise amount of bytes required for each stack.

We used these values to set the size of the application stacks, thus reducing the size of the bss section as much as possible and enabling a fair comparison. As interrupts happen non-deterministically, a single simulation run might not catch the worst case of interrupt function invocations. Thus, we added a safety margin of 20 bytes to each stack and so far no stack overflows occurred during our measurements, which of course is also monitored by the plugin.

7. RESULTS

A major observation of the evaluation is that the results are deterministic. Thus we can directly interpret these values without any additional statistics methods.

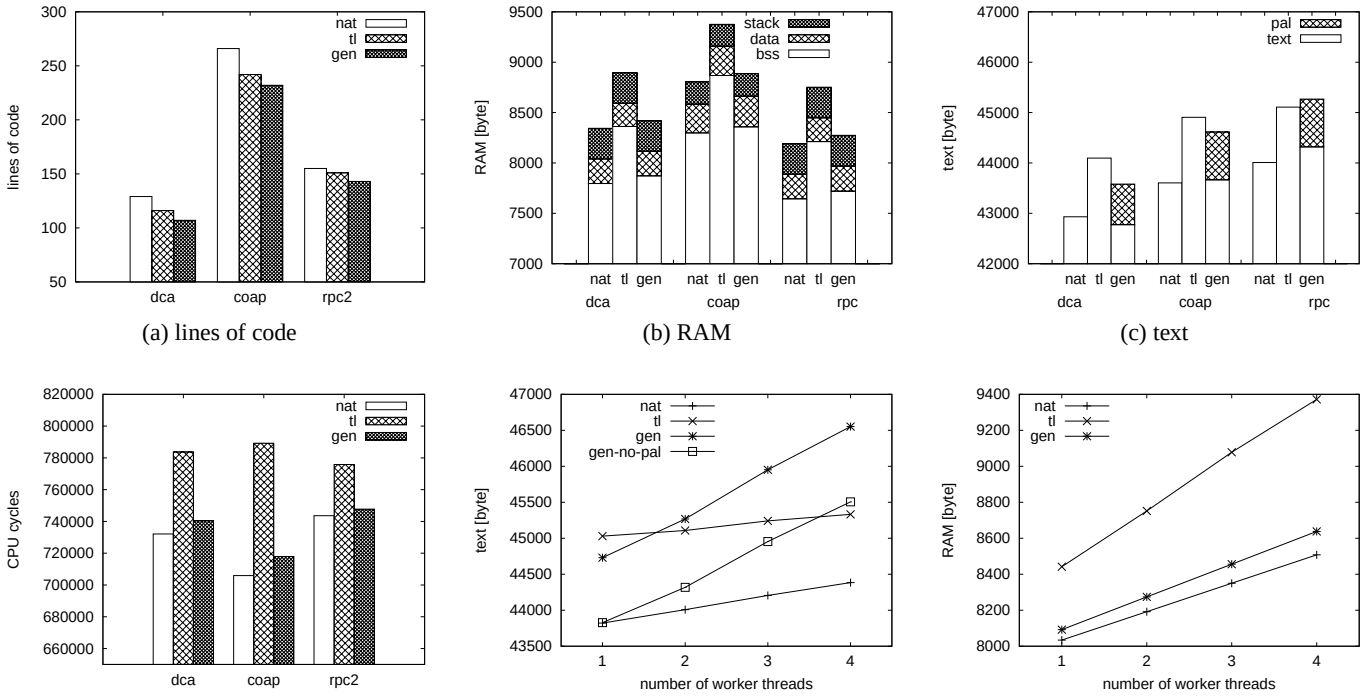
Figure 9a shows that in order to implement the same application, T-code requires 8 % to 17 % less *lines of code* than a native Contiki implementation. As already mentioned in the previous section, this measurement does not cover shared translation units, which contain additional 2000 lines of code for CoAP and 300 for RPC.

As "with protothreads the number of lines of code was reduced by one third" [6], we can estimate that a T-code application requires up to 45 % less lines of code than an equivalent event-based application. Although this result is not precise, it still supports our initial motivation for this work: synchronous functions and sequential computation provide an easier programming model than asynchronous functions and event handlers. The TL variant is close to GEN but higher because it provides the same programming model as GEN, but requires extra lines to define the application stacks and to start the threads.

7.1 Memory Resources

Of course, we expect our thread abstraction to also have some costs. First of all, we are interested in the overall memory consumption because random access memory (RAM) is very limited on sensor network devices. To this end, Figure 9b shows the size of the *data* and the *bss* section along with the maximum *stack size* for each variant of each application. First, we can see that all three variants have roughly the same amount of initialized data and a large common block of bss memory. This is because each variant uses the same operating system that adds its string constants, network stack buffers, etc.

Additionally, we can see that all three variants have roughly equal maximum stack sizes because none of them uses the runtime stack a lot: Protothreads use `static` variables, TL uses the stacks of the application threads for function-local automatic variables, and GEN has its T-stacks. As a consequence, we can see TL and GEN having larger bss segments.



(a) lines of code (b) RAM (c) text

(d) CPU cycles (e) text per thread (f) RAM per thread

Figure 9: *Evaluation results*: (a)-(d): resource consumption of various resources per variant and application. (e)-(f): resources consumption of RPC application per variant versus number of threads.

The interesting observation is that TL's overhead is significantly higher which reflects the common wisdom that thread libraries are expensive (cf. Section 3.1). A second interesting observation is that both GEN's overhead of the bss section and its overhead of the total amount of required RAM is approx. 1 % compared to NAT.

Another limited resource of sensor network devices is ROM space, which means that we need to compare the binary size of the variants. Figure 9c thus shows the size of the *text* sections and, in case of the GEN variant, it also distinguishes between code resulting from the application layer and code added by the PAL.

First, we see the overhead of TL's scheduler as expected. Similarly, we see the overhead of GEN's PAL. But we also see that the generated code itself, i.e., the E-code not including the PAL, is almost the same size as the code of the NAT variant. And including the PAL, the overhead is below 3 %.

Overall, the results show that Ocram provides the comfort of threads for just a small amount of extra RAM and ROM.

7.2 CPU cycles

Next, we want to know if the generated code involves more computation than a native implementation because keeping the CPU busy prevents the device from going into a low-power idle state. To this end, Figure 9d shows the *CPU cycles* count for each variant of each application. The absolute range of the values depends on the duration of the simulation of each application and thus provides little insight. But what we do see is that TL's scheduler adds up to 12 % of CPU cycles compared to the NAT variant. And we see that despite the additional PAL, GEN's number of CPU cycles is only approx. 2 % higher compared to NAT.

Although we do not know the exact division between the PAL and the application code, regarding the extra work performed by the PAL, this result suggests that the E-code actually requires less CPU cycles than the native implementation. An explanation for this is that all interruptable functions are inlined in the E-code, thus saving extra cycles for function prologue and epilogues. But of course, this comes at the cost of a larger binary size in case of reentrant functions.

7.3 Resource consumption per thread

To analyze this trade-off, we varied the number of worker threads of the RPC application from one to four and performed the measurements for each configuration. In each case, we adapted the client mote to send the right amount of concurrent remotely waiting RPC calls to have all worker threads busy at the same time. The previous figures showed results for the RPC application with two worker threads and using more than four workers already exceeds the limited resources of the Tmote Sky.

Figure 9e shows the size of the *text* section of all three variants versus the number of worker threads. Additionally, it shows the PAL's share by plotting the GEN variant with the text section of the PAL removed.

In the case of a single worker thread, we see the overhead of TL's scheduler once again. But we also see that TL has the lowest slope of all, as increasing the number of worker threads only involves starting another application thread. The rest of TL's code is generic in that sense and can be reused. For NAT, we have a slope of 187 bytes per worker. This originates from the additional protothreads that run the extra workers. Although they share common code, the basic stub of each protothread is always required.

And finally for GEN, we see the biggest slope of 607 bytes per worker including the PAL and 559 bytes per worker without the PAL. The share added by the PAL has the same origin as in the NAT case. And the share added by the E-code reflects the trade-off that we have chosen for the translation scheme: By inlining inter-

ruptable functions, we save CPU cycles and RAM, but we pay extra for re-entrant interruptable functions. In this sense, the thread pools of the RPC application are a worst case scenario for this translation scheme.

Figure 9f shows the other side of the trade-off which is RAM. There we see that GEN has almost the same slope as NAT, i.e., 182 vs. 158 bytes per worker. In contrast, TL has a slope of 310 bytes per worker which has two major reasons. First, as already explained, each thread's stack needs an extra margin to be able to host any occurring interrupt handler functions. And second, the generic nature of a thread library indeed saves in binary size, but it pays in RAM because it has to support all possible cases. Given the limited resources, the lack of static analysis is what kept thread libraries from improving their performance [17]. With compiler-assisted thread abstractions we aim to exploit exactly this possibility.

7.4 Discussion

Overall, the evaluation results give three major insights. First, the thread-based programming paradigm yields to more compact code compared to event-based programming, which is an indication for what is generally perceived as being an easier programming model. Second, the performance overhead of generated E-code compared to hand-written event-based code is in the single digits, making the comfort of threads affordable for resource-constrained WSN devices. And third, the performance of generated E-code is better than what can currently be achieved with thread libraries.

Of course, a dedicated threads-to-events compiler is a huge initial effort, but we argue that the major advantage of compiler-assisted approaches is their capability to exploit application-specific optimizations. Although Ocram currently only applies some basic optimizations such as keeping non-critical variables on the shared runtime stack, on the medium term we expect the performance of E-code to improve with more advanced translations, while for thread libraries we do not see such opportunities of improvement.

8. FUTURE WORK

Although we believe that Ocram represents a major step towards a practical and efficient thread abstraction for sensor networks, several improvements are subject of our ongoing and future work.

Our current compiler prototype only implements some optimizations, but there are a number of further optimization opportunities. For example, if a function is only called from one location in the whole program, there is no need to save the caller's continuation in memory. Instead, it can be hard-coded into the E-code. As a second example, in case of read-only parameters, the callee can access the variables from the caller's T-stack frame instead of having them copied to its own frame. Furthermore, different translation schemes have different trade-offs. A future T-code compiler therefore could measure properties of the input application and use a heuristic to choose the most efficient translation scheme.

We have also developed an early prototype of a T-code debugger, which offers typical source-level debugging primitives such as break points and variable inspection on T-code level. To this end, the Ocram compiler saves the performed variable and code mapping in a separate debug file and the T-code debugger communicates with a common E-code debugger such as the GDB. Whenever the user wants to access a source code line or a variable, the T-code debugger consults the debug file to find the corresponding E-code source line or variable and delegates a translated request to the E-code debugger. By these means it is possible to completely hide the event-based nature of the runtime system from the programmer. Most other WSN programming abstractions fail to support fault diagnosis, forcing the developer to cope with the complexity of the underlying run-time system nevertheless and thus breaking the abstraction.

9. CONCLUSIONS

We presented the first comprehensive compiler-assisted thread abstraction to address the increasing mote software complexity at the service and application layers, where the simple event-action model of event-based programming leads to code that is hard to manage and prone to errors.

Our evaluation showed that with our approach thread-based programming is almost as efficient as event-based programming: The overhead of RAM is approx. 1 %, for ROM below 3 %, and concerning CPU cycles the overhead is below 2 %. Thus, our work shows that the trade-off between expressiveness and efficiency of thread abstractions is not inherent to resource-constrained systems.

Additional optimizations, which are part of our ongoing and future work, are expected to stress this point even further. Thus we argue that the effort of employing a dedicated threads-to-events compiler pays off. We also believe that Ocram is a practical solution, as the generated code integrates seamlessly with existing hand-written event-based code. Furthermore, Ocram can be easily ported to other event-based kernels by implementing a thin platform abstraction layer, which is a one-time overhead.

10. ACKNOWLEDGMENTS

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.

We want to thank Matthias Kovatsch, Institute for Pervasive Computing, ETH Zurich, for helping us with his expertise in Contiki, Cooja and CoAP.

11. REFERENCES

[1] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative Task Management Without Manual Stack Management. In *Proceedings of the General Track of the USENIX Technical Conference*, pages 289–302, 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 and Applications*, 10(4):563–579, 2005.

[3] C. Bormann and Z. Shelby. Blockwise Transfers in CoAP. `http://tools.ietf.org/html/draft-ietf-core-block-10`, 2011.

[4] P. Costa, L. Mottola, A. L. Murphy, and G. P. Picco. Programming Wireless Sensor Networks with the TeenyLime Middleware. In *Proceedings of the Middleware Conference*, pages 429–449, 2007.

[5] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proceedings of the Conference on Local Computer Networks*, LCN, pages 455–462, 2004.

[6] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying Event-driven Programming of Memory-constrained Embedded Systems. In *Proceedings of*

the Conference on Embedded Networked Sensor Systems, SenSys, pages 29–42, 2006.

[7] A. Hasler, I. Talzi, J. Beutel, C. Tschudin, and S. Gruber. Wireless Sensor Networks in Permafrost Research: Concept, Requirements, Implementation, and Challenges. In *Proceedings of the Conference on Permafrost*, pages 669–674, 2008.

[8] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. *ACM SIGARCH Computer Architecture News*, 28(5):93–104, 2000.

[9] J. W. Hui and D. E. Culler. IP is Dead, Long Live IP for Wireless Sensor Networks. In *Proceedings of the Conference on Embedded Networked Sensor Systems*, SenSys, pages 15–28, 2008.

[10] K. Klues, C.-J. M. Liang, J. Paek, R. Musaloiu-E, P. Levis, A. Terzis, and R. Govindan. TOSThreads: Thread-safe and Non-invasive Preemption in TinyOS. In *Proceedings of the Conference on Embedded Networked Sensor Systems*, SenSys, pages 127–140, 2009.

[11] J. Ko, K. Klues, C. Richter, W. Hofer, B. Kusy, M. Brünig, T. Schmid, Q. Wang, P. Dutta, and A. Terzis. Low Power or High Performance? A Tradeoff Whose Time Has Come (and Nearly Gone). In *Proceedings of the European Conference on Wireless Sensor Networks*, EWSN, pages 98–114, 2012.

[12] M. Kovatsch, S. Duquennoy, and A. Dunkels. A Low-Power CoAP for Contiki. In *Proceedings of the Conference on Mobile Ad-hoc and Sensor Systems*, MASS, pages 855–860, 2011.

[13] H. C. Lauer and R. M. Needham. On the Duality of Operating System Structures. *Operating Systems Review*, 13(2):3–19, 1979.

[14] E. A. Lee. The Problem with Threads. *IEEE Computer*, 39(5):33–42, 2006.

[15] T. D. May, S. H. Dunning, and G. A. Dowding. An RPC Design for Wireless Sensor Networks. *Pervasive Computing and Communications*, 2(4):384–397, 2007.

[16] W. P. McCartney and N. Sridhar. Abstractions for Safe Concurrent Programming in Networked Embedded Systems. In *Proceedings of the Conference on Embedded Networked Sensor Systems*, SenSys, pages 167–180, 2006.

[17] W. P. McCartney and N. Sridhar. Stackless Preemptive Multi-Threading for TinyOS. In *Proceedings of the Conference on Distributed Computing in Sensor Systems*, DCOSS, pages 1–8, 2011.

[18] L. Mottola, G. P. Picco, P. Valleri, F. J. Oppermann, and K. Römer. The makeSense Programming Model. Technical Report D-3.1, Swedish Institute of Computer Science, Università degli Studi di Trento, Universität zu Lübeck, 2011.

[19] C. Nitta, R. Pandey, and Y. Ramin. Y-threads: Supporting Concurrency in Wireless Sensor Networks. In *Proceedings of the Conference on Distributed Computing in Sensor Systems*, DCOSS, pages 169–184, 2006.

[20] N. Ramsey, J. Dias, and S. Peyton Jones. Hoopl: A Modular, Reusable Library for Dataflow Analysis and Transformation. In *Proceedings of the Symposium on Haskell*, pages 121–134, 2010.

[21] J. Sallai, M. Maróti, and A. Lédeczi. A Concurrency Abstraction for Reliable Sensor Network Applications. In *Proceedings of the Conference on Reliable Systems on Unreliable Networked Platforms*, pages 143–160, 2007.

[22] Z. Shelby, K. Hartke, C. Bormann, and B. Frank. Constrained Application Protocol (CoAP). `http://tools.ietf.org/html/draft-ietf-core-coap-13`, 2011.

[23] H. Sutter and J. Larus. Software and the Concurrency Revolution. *Queue*, 3(7):54–62, 2005.

[24] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer. Capriccio: Scalable Threads for Internet Services. In *Proceedings of the Symposium on Operating Systems Principles*, SOSP, pages 268–281, 2003.

[25] M. Welsh and G. Mainland. Programming Sensor Networks Using Abstract Regions. In *Proceedings of the Symposium on Networked Systems Design and Implementation*, NSDI, pages 29–42, 2004.

[26] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. E. Culler. Marionette: Using RPC for Interactive Development and Debugging of Wireless Embedded Networks. In *Proceedings of the Conference on Information Processing in Sensor Networks*, IPSN, pages 416–423, 2006.