Institute of Operating Systems
and Computer Networks

Technische
Universität
Braunschweig

All Eyes on Code

Using Call Graphs for WSN Software Optimization

Wolf-Bastian Pöttner, Daniel Willmann, Felix Büsching, and Lars Wolf,
IEEE SenseApp, Sydney, Australia, 21/10/2013

Motivation

µDTN: Delay-tolerant Networking Implementation for Contiki

- Bundle Protocol Stack
- Network throughput was significantly lower than expected

Common Optimization Approaches

- ~~Standard profiling tools known from the PC~~
- Expert knowledge of code to "feel" bottlenecks
- Lot's of manual hacking to find bottlenecks
- Trail-and-error optimizations

→ How can WSN software be optimized in a (more) deterministic way (using standard nodes)?

Why is performance important for WSN software?

Scarce Computational Resources

- Microcontrollers are slow and speed is increasing slowly
- WSN application complexity is rising and will continue to do so (6LoWPAN, CoAP, RPL/ROLL, etc.)

Energy Consumption

- Energy supply is usually limited and scarce
- Faster execution times allow MCU to sleep longer

→ WSN Software optimization is necessary!

How to locate performance problems in WSN code?

Need knowledge of where the node spends most if its time!

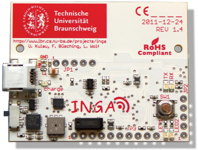
```
/* Returns a pointer to a newly allocated bundle */
struct bundle_slot_t *bundleslot_get_free()
{
    uint16_t i;
    INIT_GUARD();

    for (i=0; i<BUNDLE_NUM; i++) {
        if (bundleslots[i].ref == 0) {
            memset(&bundleslots[i], 0, sizeof(struct bundle_slot_t));

            bundleslots[i].ref++;
            bundleslots[i].type = 0;
            slots_in_use ++;

            return &bundleslots[i];
        }
    }
    return NULL;
```

+



+ ?

→ Have to instrument the code to collect information about function calls

Obtaining performance information for WSN code

Static Source Code Analysis

Does not allow conclusion on performance (in a real environment)

Instruction Set Simulators

Do not capture timing behaviour (especially of hardware components)

JTAG

Requires high read-out rate, halting the CPU and external hardware

Manual Source Code Instrumentation

Very good accuracy but does not scale

Automatic Source Code Instrumentation

Standard instrumentation does not work on microcontrollers (file I/O)

Approach in this work

1. Compiler-assisted Instrumentation of Code

Done by the GCC compiler

2. Collect Function Call Information on the Node

Using custom instrumentation functions

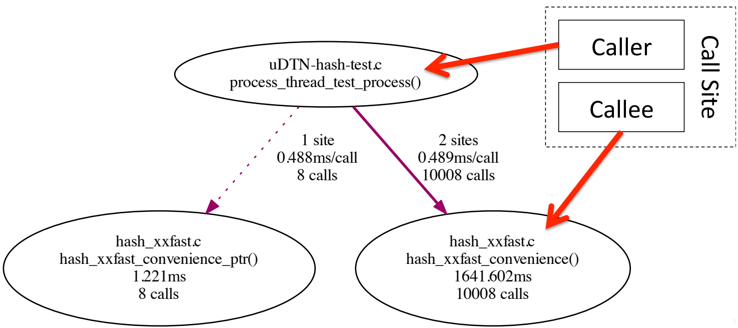
3. Transfer Collected Information to Host

On user request, off the critical path

4. Post-process and Visualize Information

Produce call graph image

Call Graphs



Instrumentation Function

Collects function call information (not part of the original user code)

1. Compiler-assisted Instrumentation of Code

- Compiler automatically modifies the intermediate code
- Inserts calls to *instrumentation functions* into each function
- Caller and Callee are provided as arguments

```
void example() {
   printf("foo");
}
```

→

```
void example() {
   profile_enter(...);
   printf("foo");
   profile_exit(...);
}
```

How to handle function call information?

Common Approach

- Transfer information about individual function calls
- **On the critical path**

What to do with function call information?

- Keep in RAM: 16 bytes per function call
- Store in flash: 0.4 ms per call (avg), 6 ms max
- Send via serial: 1.74 ms per call

→ Delay per call should be minimal, processing off the critical path!

How to handle function call information? (cont'd)

Aggregating call information on the node

- Collect information about individual function calls
- Aggregate all information regarding one call site (0.16 ms avg / call)

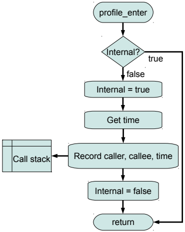
Last-In First-Out Call Stack

Sorted call site table

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31

Addr. of Caller	Addr. of Callee
Invocation Count	
Min Execution Time	Max Execution Time
Total Execution Time	

2. Collect Function Call Information on the Node

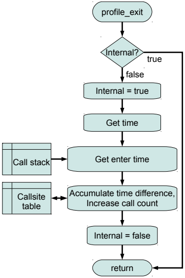


`profile_enter(void * callee, void * caller)`

- Record caller, callee and current time
- Create entry on call stack: $O(1)$

`profile_exit(void * callee, void * caller)`

- Record current time
- Obtain latest entry from call stack: $O(1)$
- Search for call site entry: $O(\log n)$
- Create / update call site entry

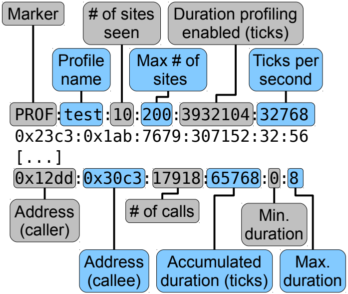


3. Transfer Collected Information to Host

Printing out call information on user request

- Can be done off the critical path
- Timing is irrelevant

Data Format

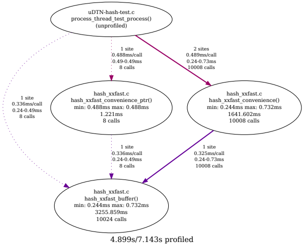


4. Post-process and Visualize Information

- Convert function addresses to function names
- Aggregate multiple call sites within a function
- Subtract execution time from outgoing function calls
- Produce image file

```
digraph G {
      label=" 4.899s/7.143s profiled";
      compound=True;
      splines=spline;
      nodesep=0.4;
      node [shape=ellipse, fontsize=10];
      edge [fontsize=9];
      process_thread_test_process [style=filled, fillcolor="#dddddd00",
      hash_xxfast_buffer [style=filled, fillcolor="#dddddd00", label="h
      hash_xxfast_convenience_ptr [style=filled, fillcolor="#dddddd00",
      hash_xxfast_convenience [style=filled, fillcolor="#dddddd00", labe
      process_thread_test_process -> hash_xxfast_convenience_ptr [color=
      hash_xxfast_convenience -> hash_xxfast_buffer [color="#650099", s
      process_thread_test_process -> hash_xxfast_convenience [color="#9
      hash_xxfast_convenience_ptr -> hash_xxfast_buffer [color="#690095"
      process_thread_test_process -> hash_xxfast_buffer [color="#690095"
}
```

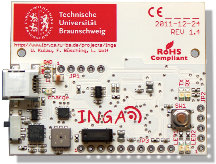
→



Implementation and Evaluation

Implementation Target Platform

- Contiki OS
- INGA and T-Mote Sky
- GCC Toolchain



→ Not limited to either Contiki or specific hardware

Evaluation Setup

- INGA
  - MCU: Atmel Atmega 1284p (128 kB ROM, 16 kB RAM, 8 MHz)
  - Radio: Atmel AT86RF231 (IEEE 802.15.4)
  - Various sensors (accelerometer, gyroscope, pressure, etc.)
- Contiki and μDTN

Evaluation Use Cases

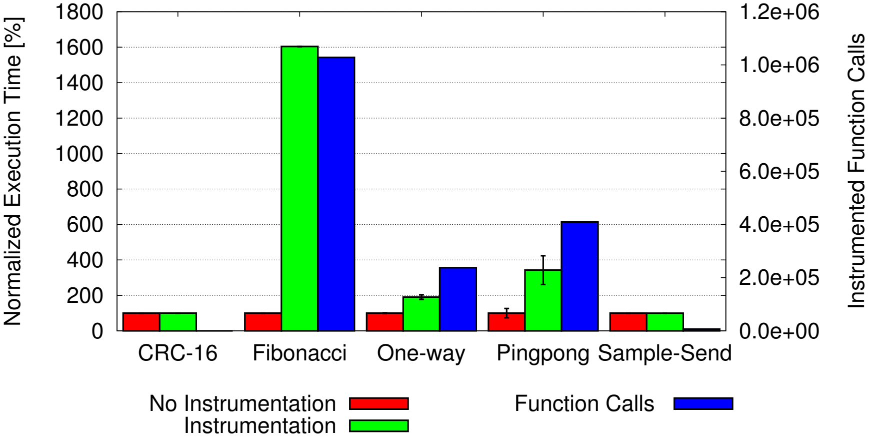
Worst-case Situations

CRC-16	Calculate CRC Checksum over 1 MB	100 % load
Fibonacci	Recursive calculation of 27 elements	100 % load
One-way	Throughput test using μDTN	100 % load
Pingpong	Roundtrip throughput test using μDTN	100 % load

Typical WSN Use Cases

Sample-Send	Typical WSN use case using μDTN	low load

Performance Implications of Instrumentation



→ Overhead strongly depends on number of function calls

RAM and ROM Overhead

RAM Overhead

- 8 bytes per call stack entry; typically 160 bytes
- 16 bytes per call site; typically 720 bytes

ROM Overhead

- 62 bytes for instrumentation functions
- 58 bytes per instrumented function
- Typical: 14 562 bytes for 250 instrumented functions

$\rightarrow$ RAM and ROM overhead is manageable on modern nodes

Conclusions

Wolf-Bastian Pöttner
poettner@ibr.cs.tu-bs.de

WSN software optimization is difficult but increasingly important

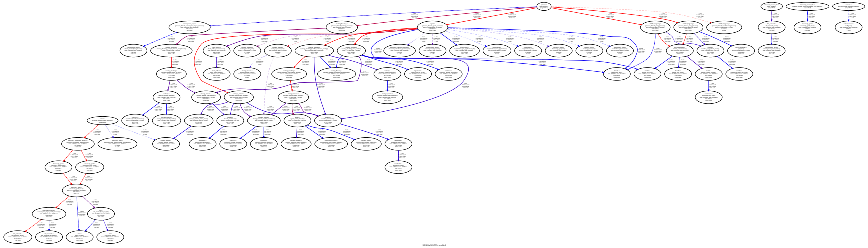
Instrumented code on nodes can be used to produce call graphs

- Call graphs allow to visually identify potential performance bottlenecks
- Code running on the nodes allows to capture the real execution environment in great detail

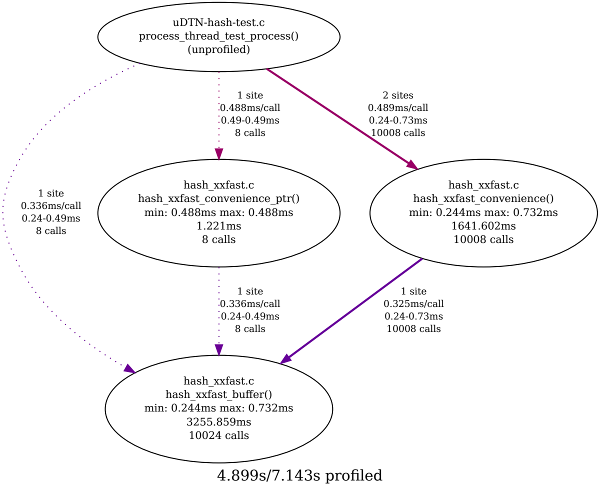
Overhead is manageable on modern nodes

- Performance impact depends on the number of function calls
- ROM and RAM overhead is manageable on modern nodes

Exemplary Call Graph 1



Exemplary Call Graph 2



Exemplary Call Graph 3