All Eyes on Code: Using Call Graphs for WSN Software Optimization

Wolf-Bastian Pöttner, Daniel Willmann, Felix Büsching, and Lars Wolf
Technische Universität Braunschweig,
Institute of Operating Systems and Computer Networks,
Mühlenpfordtstraße 23, 38106 Braunschweig, Germany
[poettner|dwill|buesch|wolf]@ibr.cs.tu-bs.de

***Abstract*—Efficient code is essential for Wireless Sensor Networks. Limited computational resources and low memory capacities require a disciplined and provident programming style. However, optimizing code requires tools to provide a deep insight into where the code may have potential for improvement. In this paper we present a way of generating call graphs of software for standard Wireless Sensor Nodes. We execute the software on the actual nodes to collect profiling information and visualize this data on a PC-based host system. The call graphs are enriched with information about function execution time, execution count and visualize the call chain of the program to allow the programmer to identify room for optimization.**

I. Introduction

Wireless Sensor Nodes are used in a multitude of applications in which the energy consumption must be as low as possible to maximize lifetime of nodes or networks. While the development of processors for PCs follows Moore's law and more powerful hardware becomes available over time, the hardware development for sensor nodes is largely focused on energy efficiency and will continue to do so for the foreseeable future.

On the other hand, applications for Wireless Sensor Networks (WSNs) are becoming more demanding already today and more so in the future. This creates a major challenge for application developers: Code must be very efficient to make best use of the scarce computational resources. Also, more efficient programs finish faster and allow the Microcontroller Unit (MCU) to sleep longer, thereby reducing the energy consumption. However, our experience in working with WSNs has shown that debugging capabilities of today's systems are limited and that performance optimization requires in-depth expert knowledge of the software that shall be optimized.

In this paper, we present an approach to generate call graphs of code running on live sensor nodes. The call graphs are enriched with additional information to provide the programmer with a quick insight into the structure of the program. Furthermore, the call graphs enable the programmer to pin point hot spots and figure out where the node spends most of its time. Overall, we present an approach to instrument code running on off-the-shelf nodes and evaluate the overhead induced by instrumenting code. Therefore, the main contributions of this paper are:

- Design and implementation of a call graph generation framework running on live WSN nodes based on compiler-assisted source code instrumentation
- Evaluation of the performance implications and accuracy of said framework

The remainder of this paper is structured as follows: In the following Section II we outline suitable performance metrics and introduce the concept of call graphs. We discuss related research efforts in Section III and present the design of our approach in Section IV. Our implementation on actual nodes is described in Section V and evaluated in Section VI. Finally, Section VII concludes the paper.

II. Basics

In this section we first clarify the terminology used in this paper. We then introduce performance metrics for an instrumented program and explain the concept of call graphs.

A. Terminology

We use the term *caller* to identify a function that is calling another function. More precisely, a caller is the instruction calling another function. One function may call another function from multiple different instructions, thus representing multiple callers. The called function is referred to as *callee.* We use the term *call site* to identify a combination of a caller and a callee. We further refer to *instrumentation functions* and *profiling functions* as functions that are called to allow instrumentation of the user code. Those functions are not part of the original code that the user wants to compile. With *source code function* we refer to the user provided source code. We use the terms programmer and user interchangeably for the creator of the source code.

B. Performance Metrics

The *performance* of a program is related to its speed. The speed in turn is related to the execution time of the whole program, which is the accumulated execution time of all functions that are involved in the program. Therefore, we can judge about the performance of a function by looking at its *execution time.* By reducing the execution time of individual functions the execution time of the whole program can be reduced. However, if said function is only called once in the program, the potential impact on the program execution time may be only marginal. Therefore, the *execution count* is another metric that is necessary to estimate the impact of the performance of a single function onto the whole program.

978-1-4799-0540-9/13/$31.00 ©2013 IEEE

A function that has a long execution time may be a good target for optimization, but only if the number of calls is significant. On the other hand, a function that consumes only little time but is called often may also be a good starting point for optimization purposes. Since one function may be called from various points in the program, execution time for each of these call sites should be recorded. This is especially important for functions that expose different execution times for different arguments. Even though a function may be called from only one site, the variance of the execution time is also interesting for the programmer. A function that exposes a high variance may be very sensitive to specific arguments. A way to express the variance in execution time is to show the minimum and maximum execution times of a function.

C. Call Graphs

A call graph is a directed graph that represents calling relationships between functions of a program. In the graph, nodes represent functions and directed edges represent function calls from one function to another function. A simple call graph can be seen in Figure 1 in which the function *process_thread_profiler()* calls function *fib()* which calls itself recursively 21,852 times. In the nodes of the graph (functions), we list the source code file and the function name as well as the minimum, maximum and cumulative execution time together with the number of calls. For each edge (function call) we list the number of call sites that are aggregated here as well as the minimum, maximum and average execution time and the number of calls. Multiple *sites* on an edge indicate that the caller has multiple invocations of the callee in different instructions.

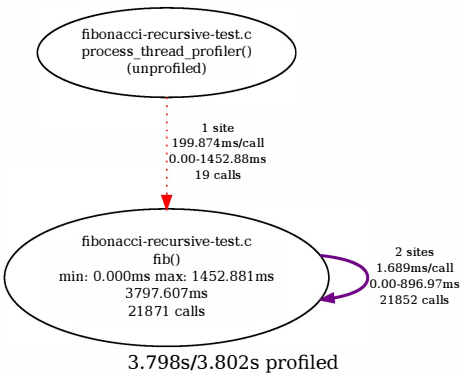


Fig. 1: Call Graph Example

With the information included in a call graph, a programmer gets a visual representation of the program flow. Calling relationships between functions as well as call chains are easily visible. With this information, the programmer can understand where the program spends most of the time and thus consumes energy.

III. Related Work

Code Profiling has been around in software engineering for a long time. In general, we can distinguish between different goals of code profiling. On the one hand, profiling the memory consumption of code is common. It allows to find bugs that lead to memory leaks and to assess the actual requirements of code towards dynamically allocatable memory. On the other hand, code execution profiling allows assessing the performance of program code. It allows figuring out how long code actually takes to complete certain operations and is a valuable basis for optimization.

Since dynamic memory allocation suffers from overhead and fragmentation problems, most WSN operating systems and applications only rarely make use of it. Contiki [1] and TinyOS [2] both offer their own means of fragmentation-free memory allocation that include debugging features and are easily expandable with profiling features. Apart from this easy-to-instrument fragmentation-free memory allocation, all memory on nodes is allocated statically at compile time. Standard tools from the GNU Compiler Collection (GCC) [3] toolchain such as *nm* and *size* allow to assess the static memory consumption. Instrumenting and profiling memory consumption on actual nodes is not necessary. Hence, the remainder of this paper focuses on execution time profiling.

Static Source Code Analysis [4] is based on the analysis of the source code without actually running the program. This technique is useful to find programming errors and potential security flaws, but is not very helpful to reach conclusions on the performance of software. Call graphs can be created[1] but do not allow counting invocations or recording execution times.

Instruction Set Simulators [5] replicate a whole MCU and allow to run executables in a simulated environment. Simulators allow instrumenting the environment of the executable to assess its performance and allow a close monitoring of what the binary is actually doing. A major advantage is that an unmodified binary can be used and that the binary cannot tell if it is being profiled. A significant drawback is the fact that timing behavior, hardware specifics and interaction can hardly be simulated accurately in such simulators, as simulation always differs from ”the real world”. Additionally, not every node can be adequately simulated due to the lack of implementations.

JTAG [6] is a debugging interface that allows to attach external debuggers to the MCU. Dependent on the specific implementation, the debugger can read all registers of the controller, including the Program Counter that contains the instruction that is currently being executed. Given a read out rate that is high enough, this would allow a close monitoring of the executed program. However, Atmel states in [7] that ”the Program Counter can not be read while the emulator is in Run Mode”. This means that program execution would have to be interrupted each time when reading the program counter which is not acceptable especially when, e.g., taking network communication into account and not only observing single nodes, but the behavior of whole networks.

Statistical Sampling [8] uses a low-level routine to periodically record which instruction the processor is executing at the moment. After mapping the recorded instruction onto actual lines in the program code, a distribution graph can be produced that shows which function has been seen how often. This method can point to a hot spot inside a function but is unable to generate call graphs and can only give approximations of

[1] http://www.gson.org/egypt/

function execution times. E.g. the GCC compiler allows to instrument binaries with statistical sampling and *gprof* [9] can interpret the results. However, the present version cannot be used in MCUs since this approach expects file IO to behave like on a regular PC.

Manual Source Code Instrumentation [10] is what many programmers use today to debug and profile code running on WSN nodes. By manually adding instructions to the code that measure the number of invocations of a function or the time spent between two points, the programmer can debug small portions of the code. While this approach allows to exactly pinpoint where the time is spent by measuring between two arbitrary points, it does not scale. Also, manual source code instrumentation requires significant manual effort and specific knowledge of the code and is prone to errors made by the programmer.

Automatic Source Code Instrumentation [11] helps the programmer by automatically adding the profiling functions to the code. During code execution, the mechanism records profiling data and writes the results to a file. E.g. GCC allows to automatically instrument the binary and *gcov*[2] can handle the result. While this approach offers a good coverage of the code, the profiling functions are provided by the compiler and cannot be modified for the target architecture. Again, File IO is expected to work as on regular PCs which neglects the use on MCUs.

Compiler-assisted Source Code Instrumentation [12] automatically inserts calls to instrumentation functions into the program code but allows the programmer to implement those functions. E.g. GCC calls separate instrumentation functions when entering and leaving a source code function and passes arguments that allow to identify the caller and the callee. The instrumentation functions can be specially crafted for the target architecture, including MCUs. The disadvantage is that this only allows to profile on function level and does not allow to dive deeper into specific functions.

TinyAID [13] is an effort for automated instrumentation of TinyOS programs allowing message and call-chain logging. However, the presented paper is limited to TinyOS and does not allow extracting performance metrics, so that the starting point for performance optimization is less clear compared to our more general approach.

For microcontroller platforms, the compiler-assisted source code instrumentation presents the best trade-off between feasibility (implementable on MCUs) and flexibility (function level profiling). To the best of our knowledge, the generic, OS independent generation of profiling data on WSN nodes is a novel concept and has not been published before.

IV. Design

Producing a call graph as introduced in Section II-C requires knowledge of function calls that are performed in the program. For each function call, the caller and the callee have to be recorded. Furthermore, the profiling metrics (see Section II-B) such as execution time (accumulated, minimum and maximum) and execution count of a function have to be stored. Based on this information, a call graph can be created.

[2]http://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[3]addr2line is part of GNU Binutils

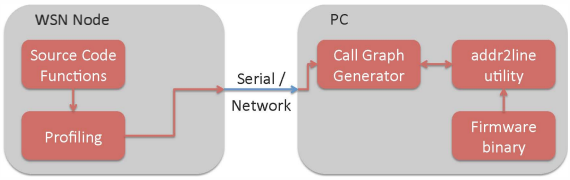


Fig. 2: Profiling Architecture[3]

On the nodes, we collect information about call sites. The actual generation of the call graph as well as post processing steps can be done on a PC-based host system. To facilitate this, the collected profiling information has to be transported to the PC as indicated in Figure 2.

A. Prerequisites

To enable compiler-assisted source code instrumentation, our approach requires a compiler that can automatically insert calls to instrumentation functions into each source code function as shown in Figure 3. The compiler inserts a call to the *enter* function as the first instruction of each source code function. Furthermore, the compiler adds a call to the *exit* function prior to leaving the function. This allows recording the instance at which each called source code function has been invoked. The approach further requires that the profiling functions are able to obtain the addresses of the caller and the callee. For this purpose we use the GCC [12] compiler suite.

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

(a) Source Code Function

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

(b) Compiler-Instrumented Function

Fig. 3: Relationship between source code and compiler-instrumented function.

B. Storing Function Call Information

To be able to draw a call graph, we need information about function calls. This information has to be collected on the node and subsequently transported to a host PC for call graph generation. For each call, we need the address of the caller and callee as well as the execution time. Since calculating the execution time on the nodes requires a call stack, a simple approach is to timestamp occasions at which functions are entered or left. We refer to this information as a *call record*. While the addresses are 2 bytes each, the timestamp has to be measured with acceptable resolution. With a resolution of 0.1 ms, 4 bytes would last 4.97 days of execution time which should be enough for most use cases. Furthermore we need type information (function enter or exit) if 1 byte and a delimiter of 1 byte. So for each function call, two call records are generated that consume 10 bytes each.

Existing approaches print out call records on the standard output. For WSN nodes this will usually be a serial connection to a host computer. On a serial connection with 115 200 bit/s and 1 start and stop bit per byte, printing one call record would

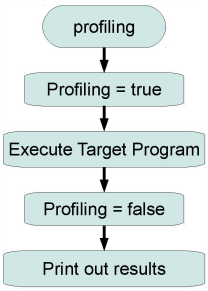


Fig. 4: Profiling Procedure

take 0.86 ms for the data to be transported off the node. Since two call records are created per function call (on entering and leaving a function), this would create delays of at least 1.74 ms per call. Therefore, using the serial port to output call records is unacceptable while the program is being profiled.

Storing call records in FLASH memory is another alternative. One flash page of the Atmel AT45DB161 serial data flash [14] holds 528 bytes of user data and allows to store up to 52 call records. With program times between 3 ms to 6 ms per flash page, this would result in a delay of 0.11 ms per call record or 0.23 ms per function call. Other serial flash chips such as the M25P80 [15] take between 1.5 ms to 5 ms to program a page of 256 bytes. This would result in a delay of 0.4 ms per source code function call. While the delays induced by writing flash memory are smaller than the delay penalty on the serial port, the delay is still high enough to interfere with the program.

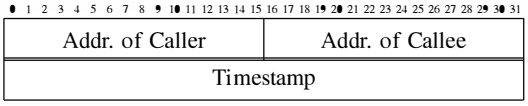
Furthermore, data for each flash page would have to be buffered in memory and then flushed once the buffer is full. This means that the delay would not be evenly distributed per function call but would aggregate at the point where the buffer is full. Therefore, writing call records to flash would expose highly variable delays per function call which is not desirable.

Thus, printing each individual function call or saving it to flash is not feasible. Instead, the call records have to be stored on the nodes in Random-access Memory (RAM). Since RAM is limited on nodes, we preprocess information about function calls by aggregating all information related to one call site. Whenever the programmer decides that it is safe, he calls a function to send the aggregated information via the network interface or the serial port (UART) as indicated in Figure 4. Different to writing pages in flash memory which has to be done every 26 function call, the information in RAM can be printed after the profiled program has ended as indicated in Figure 4. This takes the time-consuming operation (storing or printing the profiling information) off the critical path.

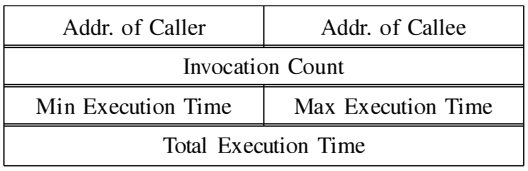
C. Instrumentation Functions

As mentioned earlier, the *instrumentation functions* are functions for which calls are inserted into the source code by the compiler when entering and leaving a source code function.

The *enter* function records the address of the caller, the callee and the current time in a data structure as shown in Figure 5a. The *exit* function searches for this entry, calculates the execution time and inserts or updates the information in



(a) Call Stack Entry

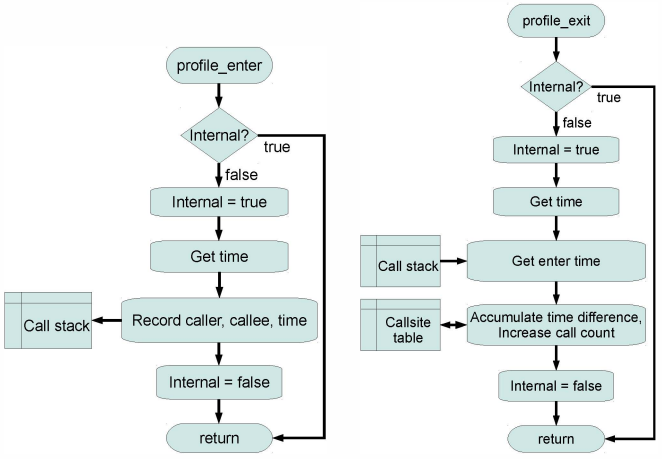


(b) Call Site Table Entry

Fig. 5: Memory structures to save the aggregated profiling data

a call site table. A single entry of this table is shown in Figure 5b. Since we know already that speed is important, we have adopted the concept of a Last-in First-out (LIFO) call stack. The *enter* function adds one entry to the call stack and the *exit* function can simply retrieve the latest entry. When implemented as a static array with a pointer that points to the last element that has been added, the access to the LIFO is executable in constant time $O(1)$.

Figure 6a shows our *enter* function. It records the current time and creates another entry on the call stack. Calling an instrumented function from within the instrumentation functions would cause in infinite recursion. To avoid this, we use a mutex called *Internal* that avoids profiling all functions that are called from within the profiling functions. Note that this does not interfere with recursive source code functions that are profiled as expected.



(a) Enter function (b) Exit function

Fig. 6: Instrumentation Functions

Similarly, Figure 6b shows the *exit* function. It records the current time and retrieves the latest element from the call stack. The next step is to calculate the execution time and find the appropriate entry in the table of call sites. Once found, the data regarding this call site is updated. We use the same mutex to prevent infinite recursion.

Finding the proper call site in the table is time critical and

has to be fast. By using a binary search algorithm, the runtime complexity is $O(\log n)$.

The memory consumption of a typical call stack with 20 elements (allowing function call chains of 20 functions) consumes 8 bytes per entry of 160 bytes in total. Each profiling site consumes 16 bytes each, so a typical table with 45 call sites consumes 720 bytes of RAM.

D. Further Processing Steps

On the PC, function addresses can be converted back to function names by using the compiled binary file with debugging information. This allows to make the collected call site information human readable. The next step is to create a list of nodes (functions) and a list of edges (function calls). That information can be expressed in a language for specifying graphs. One example is the Graphviz DOT language [16] for which tools for inspection and rendering exist.

One important element of the post processing steps is to alter the execution time of all functions. In the call graph, we want to show the actual execution time of a function in the nodes. However, if this function calls other functions, the time spent in those called functions shall not be attributed to the calling function. Therefore, we subtract the execution time of all outgoing function calls from the execution time of each node.

V. Implementation

Our approach does not require any special hardware; it only has to be supported by a GCC compiler. We have implemented our approach for Contiki OS [1] running on the INGA [17] hardware platform. INGA is based on the Atmel ATmega1284P microcontroller and the Atmel AT86RF231 radio transceiver. We have used the GNU open-source toolchain consisting of GCC[4], AVR C Library[5] and AVR binutils[6]. The implementation presented here is open-source software and available from our GIT repository[7].

We have implemented the profiling architecture as part of Contiki OS, so that minimal modification of user code is necessary. To use our approach, the user has to change the compiler flags (CFLAGS) to configure the compiler to enable instrumentation. To instrument a subset of the source files or a subset of the functions of a source file, additional options can be passed to the compiler. Furthermore, the user has to call a function to print out the profiling results onto the serial port or to send it over the network once the to-be-profiled code is finished. On the PC we have implemented a python framework that compiles source code, flashes the nodes, collects the profiling information and automatically creates the call graph in PDF format.

Contiki allows processes to yield on order to allow other processes to execute. Since processes in Contiki are actually functions, yielding a process calls return and when the scheduler decides to resume the process, another call to the function of the process is issued. Therefore, our profiling approach sees a yielding process as two function calls to the same function of the process.

A. Accuracy and Resolution of Time Measurement

Measuring the precise execution time of functions is important for accurately profiling a program. The actual accuracy of a clock source depends on a number of factors. However, the granularity may also become a limiting factor. If the granularity of the timing source is too coarse, short-running functions may not have any execution time attributed to them in the call graph. On the INGA hardware platform, the function *clock_fine()* provides time with a resolution of 4096 ticks per second or 0.244 ms per tick. This level of granularity should be enough for most profiling use cases. We evaluate the accuracy of time measurement on a specific platform in Section VI-G.

B. Instrumenting Library Functions

As outlined in Section IV-A, compiler-assisted instrumentation automatically instruments functions compiled from source code. However, libraries are usually present as a binary version and are linked into the final binary without being compiled each time. Therefore, those functions are also not part of the instrumentation. However, the C library contains many functions that are relevant for performance such as *memcpy()*. Writing custom version of the library functions is possible, but would require changing all calls to such functions in the source code which is not desirable.

The GCC compiler offers a way to replace calls to certain library functions with a call to a wrapper function without the need of modifying the code. Those wrapper functions are part of the source code and are therefore also instrumented by the compiler and call the original library functions. This allows profiling arbitrary existing library functions with minimal overhead without changing the function calls in the source code.

C. Problems with Inlining

A common performance optimization of compilers called inlining is to embed code of certain functions into the caller instead of performing the actual function call. For simple functions, this approach reduces the overhead of function calls by reducing its number. Unfortunately, GCC handles instrumentation of inlined functions the wrong way; the corresponding bug ticket is open since 2005[8]. GCC erroneously calls the instrumentation functions for the inlined functions with parameters of the caller (instead of the callee).

For functions *a()* calling *b()* and *b()* calling *c()* the compiler may inline *c()* into *b()*. The instrumentation functions are now called two times, each time indicating a call from *a()* to *b()*. The second call is erroneous, since in fact *c()* was called by *b()*. This leads to wrong execution counts and wrong execution times for functions *b()* and *c()*. The best workaround we could find is to disable inlining at all, using a compiler option until the GCC bug is fixed. However, this solution goes at the expense of performance as we show in the evaluation Section VI.

[4] http://gcc.gnu.org/

[5] http://savannah.nongnu.org/projects/avr-libc/

[6] http://www.gnu.org/software/binutils/

[7] http://git.ibr.cs.tu-bs.de/?p=project-cm-2012-inga-contiki.git

[8] http://gcc.gnu.org/bugzilla/show_bug.cgi?id=23296 and http://gcc.gnu.org/bugzilla/show_bug.cgi?id=28205

VI. Evaluation

The goal of the evaluation is to examine the impact of our instrumentation approach onto the software running on the nodes. Furthermore, we want to evaluate the positive impact that our approach can make. Since we cannot measure this in an objective way, we present the optimization of a networking stack for Contiki as an example for how code can be optimized using call graph information. We furthermore present five common WSN tasks and measure the impact of instrumentation and inlining on the performance and Read-only Memory (ROM) consumption. This creates an idea of what overhead to expect when instrumenting code. All following measurements are based on at least 50 experiment runs on actual nodes in our university lab. We present the arithmetic mean as well as the standard deviation.

A. Example: Optimizing μDTN

μDTN [18] is a Bundle Protocol implementation for Contiki OS. It can be used to overcome situations with intermittently connected nodes by transporting data in bundles that can be temporarily stored in nodes. Performance comparison with uIP [19] have revealed that the application-layer throughput of μDTN was significantly slower [20]. The throughput of μDTN was 2963 bytes/s whereas uIP achieves 10 204 bytes/s. We have generated a call graph for this test case and found that in the send path of μDTN the *mmem_realloc()* function was called very often and consumed significant amounts of time. Upon inspection of the code we found that more and more memory was incrementally allocated for each of the 19 header fields, each time calling *mmem_realloc()*. Each of these invocations involves several calls to *memcpy()*. We have restructured the code of μDTN based on the information shown in the call graph and achieved a throughput of 5947.5 bytes/s. We found the call graph to be quite handy to understand how complex programs (such as μDTN) work and to visually comprehend where the time is spent. An excerpt of an exemplary call graph is shown in Figure 7; the full figure can be found in the μDTN Wiki[9].

B. Measurement Methodology

To evaluate the performance impact of our instrumentation approach, we measure the execution time of five tasks on a sensor node. We have selected the five tasks to cover the typical areas in which sensor nodes operate: computation (recursive and non-recursive) and networking (interactive and non-interactive). We furthermore investigate tasks limited by the available computational resources as well as a typical WSNs use-case with periodic sampling that contains sleep periods. In all tests we have instrumented the user program as well as μDTN (if used) but did not instrument the underlying Contiki operating system.

1) *CRC-16*: The rationale behind this scenario is to imitate a computationally intensive task on a node. We calculate the CRC-16 checksum over 1 Mb of arbitrary data. The test is implemented as a single function and uses all available computational resources.
2) *Fibonacci*: This scenario is also computationally intensive and uses an extensive amount of function calls. We calculate the first 27 numbers of the Fibonacci sequence. The test is implemented in a recursive function in which each call will produce two additional calls. The recursion stops when the initial Fibonacci values are reached.
3) *One-way*: One-way is a typical one-way networking application. We use μDTN in a simple throughput test in which one node is a sender and generates data as fast as possible, thereby using all computational resources. Data is transmitted to a receiver and consumed by an application running there. We measure the time until 1000 bundles of 80 bytes payload have been transmitted.
4) *Pingpong*: Pingpong is an interactive networking application. Again we use μDTN with two nodes in which the sender sends a packet to a receiver. The receiver replies with the same bundle. We measure the time until 1000 bundles of 80 bytes payload have been echoed by the receiver. This scenario uses all available computational resources because bundles are sent as fast as possible.
5) *Sample-Send*: Sample-Send is a typical WSN application in which the sender samples a sensor at 1 Hz and sends each sample in a μDTN bundle to the receiver. We measure the time until 60 bundles have been received by the receiver. This scenario is limited by the sample rate and uses far from all computational resources.

[9] http://trac.ibr.cs.tu-bs.de/project-cm-2012-mudtn/wiki/PerformanceOptimization

C. Performance Implications of Instrumentation

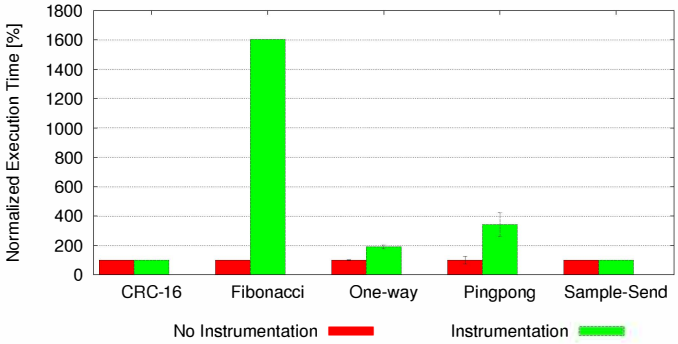


Fig. 8: Execution time comparison for code with and without source code instrumentation (lower is better).

Figure 8 compares the performance of our five tasks with instrumentation enabled and disabled. We furthermore show the number of instrumented functions, the instrumented function calls and their standard deviation in Table I. Regarding performance we see that the CRC test does not suffer from any performance degradation. This was expected since it only issues a single function call. The other CPU intensive tests (Fibonacci, One-way, Pingpong) suffer from increased execution time (decreased performance). As expected, the impact on performance correlates with the number of function calls. The typical WSN use case (Sample-Send) does not suffer from any negative performance impact.

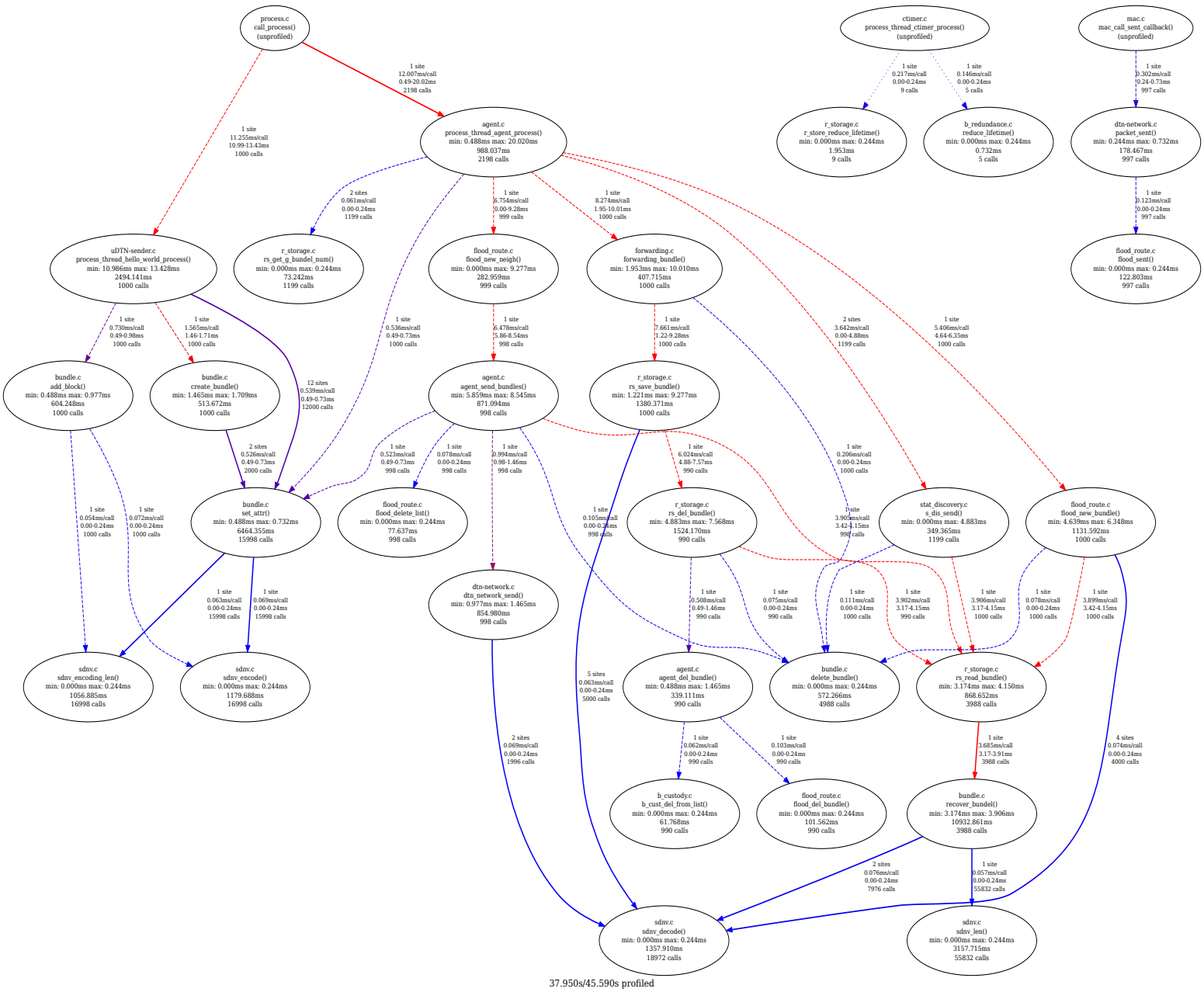


Fig. 7: Excerpt of an exemplary call graph taken with μDTN

	Instr. Fct.	No. of Function Calls	Standard Deviation
CRC	2	1	0 %
Fibonacci	2	1028309	0 %
One-way	218	409262	6.03 %
Pingpong	221	409262	0.04 %
Sample-Send	218	6940	0.06 %

TABLE I: Number of instrumented function calls and the standard deviation for the five tasks.

Although the performance impact on the Fibonacci example is devastating, this is a task that we rarely see in practice. Even if programs contain recursive functions, it is unlikely that those are the only functions in the program. Thus issuing 1,028,309 calls in this test is the worst-case behavior and the performance degradation in practice will be less severe. The CPU intensive networking tasks (One-way and Pingpong) also experience significantly increased execution time and thus similarly decreased throughput and increased latency. However, μDTN is still working as intended and even interaction with other nodes works. For the typical WSN use case (Sample-Send) performance is not influenced because for a sample rate of 1 Hz the node sleeps most of the time. Enabling instrumentation reduces the sleep periods but does not influence overall performance.

The results show that especially CPU intensive tasks are heavily impacted by enabling instrumentation. Furthermore, the performance degradation correlates with the number of instrumented function calls. However, regular operation of those programs is not interfered with. For typical use cases that are not bounded by the computational resources, profiling does not have a performance impact independent of the number of instrumented function calls. For use cases that need all available computational resources, instrumentation may interfere with the intended operation of the program. It may be necessary to instrument only a subset of the functions to reduce the performance impact and to restore the intended operation of the program.

D. Performance Implications of Inlining

As outlined in Section V-C, we have to disable the inlining compiler optimization to work around a bug in GCC. In Figure 9 we show the execution time of four use cases with function inlining enabled and disabled. The execution time is normalized to the the test case with inlining enabled (that is the default for GCC). We see that for the CRC and Fibonacci experiments, inlining does not make a difference. This was expected, since the respective functions are either called recursively (and inlining is not possible) or the functions are too complex to inline them. For the One-way experiment, disabling inlining actually increases performance slightly. However, the increase is well within the standard deviation and therefore not significant. In the Pingpong test the performance is decreased slightly when disabling inlining which was expected. We learn that disabling inlining does not have a significant performance impact for most of our use cases.

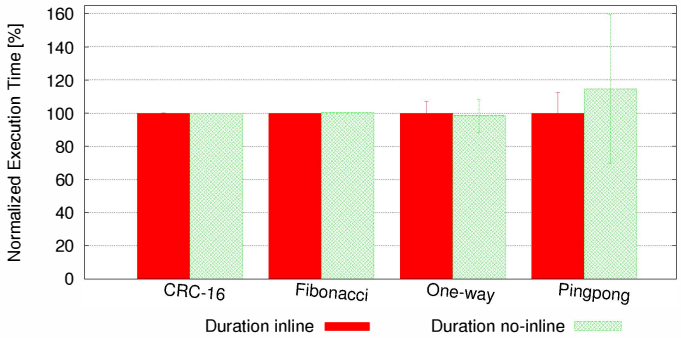


Fig. 9: Execution time comparison for code with and without function inlining. (lower is better)

E. Overhead per Source Code Function Call

In Section IV-B we argue that the instrumentation functions have to be fast to avoid disturbing the user program too much. We look at the Fibonacci task to figure out the timing overhead of calling the instrumentation functions. We use this task, because it does the most function calls which yields the highest accuracy for this analysis. We have divided the total execution time difference between instrumented and non-instrumented execution by the number of function calls. We see that the time overhead per source code function call is $162.9\,\mu$s.

Calling the instrumentation functions consumes time for two reasons: On the one hand, the body of the profiling functions has to execute and that takes time. On the other hand, performing the function call itself involves several operations that also cost time. We have measured the execution time of the Fibonacci use case without any instrumentation and with instrumentation functions that only contain a single operation. The simple instrumentation functions increase the execution time by $11.6\,\mu$s per source code function call. This means that the body of our implementation of the regular instrumentation functions takes $151.3\,\mu$s to execute.

	Fct. Call	Fct. Body	Total
On-Node Aggregation	$11.6\,\mu$s	$151.3\,\mu$s	$162.9\,\mu$s
Call records via serial 19 200 bit/s	$11.6\,\mu$s	$10\,416.7\,\mu$s	$10\,428.3\,\mu$s
Call records via serial 115 200 bit/s	$11.6\,\mu$s	$1736.1\,\mu$s	$1747.1\,\mu$s
Call records to Flash (AT45DB161)	$11.6\,\mu$s	$230.8\,\mu$s	$242.4\,\mu$s

TABLE II: Execution time increase per source code functions in comparison to alternative ways of storing the instrumentation data.

We compare to the three alternative approaches of recording instrumentation data (see Section IV-B) in Table II and see that our approach (On-Node Aggregation) is significantly faster per function call. Since the other approaches have to transport more information either over the serial port or into flash during each function call, aggregating call site information on the node saves precious time. The nearest competitor to our approach is flash memory that can only be programmed in pages and would expose a highly variable delay as explained in Section IV-B.

F. ROM Overhead

Instrumenting a program produces a larger binary program because calls to the instrumentation functions must be inserted in each instrumented source code function. Also, code is statically appended to the binary program for the instrumentation functions. We have compiled a sample program with 250 functions. We compare the ROM size of the program without instrumentation and then gradually enable instrumentation for one function after the other. We have found, that the ROM size is increased by 62 bytes of static overhead. This overhead is caused by the instrumentation functions. Furthermore, each instrumented function increases the ROM size by 58 bytes because the calls to the instrumentation functions are inserted. In total we have found, that a program with 250 instrumented functions shows an increased size of 14 562 bytes compared to the uninstrumented version. Compared to the 128 KBytes ROM of INGA, this overhead is manageable.

G. Timing accuracy

Since we measure the execution time of functions, we want to figure out how accurate this time measurement really is. For this purpose, we created a simple program that toggles an IO pin of the MCU. We ran this program with instrumentation and created a call graph. Furthermore, we have sampled the pin at 16 MHz with a logic analyzer and measured the time in a specific state. The ideal result would be that the duration recorded by the instrumentation and by the logic analyzer are the same.

Multiple measurements with the logic analyzer show that the call takes 1.435 ms on average. With the instrumentation functions we measure an execution time of 1.465 ms. Thus,

the inaccuracy of time measurement is in the order of 0.03 ms which should be good enough for most applications.

VII. Conclusion

Optimizing code running on MCUs for maximum performance is troublesome and requires expert knowledge. The primary reason for this is that existing debugging and profiling tools usually cannot be used on the MCU as these tools are optimized to be used on PCs. Existing simulation approaches allow in-depth instrumentation of the actual code, but only run the code in a simulated environment with unclear consequences and constraints especially when it comes to network interaction and peripherals such as flash memory or sensors.

Our tools, which are available from our GIT repository (cf. Section V), help developers by instrumenting code running on real wireless sensor nodes. We collect information about function calls and pre-aggregate this information in the nodes RAM. By avoiding to store or print information about each individual call record on the critical path (during a function call), our approach saves precious time and has a lower timing overhead than alternative approaches. The programmer can decide when it is safe (from a timing perspective) to send the aggregated data to a PC via serial connection or via the wireless network. On the PC, the data is interpreted and a call graph is created that allows the programmer to understand the flow of the program and to identify hot spots that are worth to optimize.

The evaluation has shown that instrumenting code produces an average delay of 162.9 μs per source code function call. Compared to alternative ways (storing data in flash memory) of handling the collected data, this is a decrease in timing overhead of 32.8 %. The execution time if tasks are limited by the available computational resources depends on the number of function calls and can be between severe (Fibonacci with many recursive function calls) and modest (One-way, a throughput task involving network transfers). The results further show that a typical WSN use case in which data is sampled every second and send to another node suffers no performance degradation because the node spends most of the time in idle mode and the computational resources are not the limiting factor. The overhead in terms of RAM and ROM is a 68 bytes larger binary for each instrumented source code function and a 175 bytes larger binary for our implementation of the instrumentation functions. Also, 16 bytes of additional RAM consumption for each call site and 8 bytes per entry on the call stack.

We used this approach to optimize μDTN, a bundle protocol implementation for Contiki OS. In our experience, looking at the visual representation of the flow of a program is a good starting point for further optimization. We were able to increased the networking throughput of μDTN by 100.7 %.

All in all, the authors claim that the use of call graphs for WSN software optimization should be obvious by now. Whenever there is a demand for optimization, simply instrument the code and study the automatically generated call graphs.

References

[1] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors," in *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, Nov. 2004.

[2] P. Levis and D. Gay, *TinyOS Programming*, 1st ed. New York, NY, USA: Cambridge University Press, 2009.

[3] Free Software Foundation, Inc., "Using the GNU Compiler Collection," http://gcc.gnu.org/onlinedocs/gcc/, [accessed 11-Dec-2012].

[4] L. Mottola, T. Voigt, F. Österlind, J. Eriksson, L. Baresi, and C. Ghezzi, "Anquiro: enabling efficient static verification of sensor network software," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Sensor Network Applications*, ser. SESENA '10. New York, NY, USA: ACM, 2010, pp. 32–37.

[5] B. L. Titzer and J. Palsberg, "Nonintrusive precision instrumentation of microcontroller software," in *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, ser. LCTES '05. New York, NY, USA: ACM, 2005, pp. 59–68.

[6] IEEE Computer Society (Test Technology Standards Committee). (2001, Oct.) IEEE 1149.1-2001 Standard Test Access Port and Boundary-Scan Architecture.

[7] Atmel Corporation, "AVR060: JTAG ICE Communication Protocol," http://www.atmel.com/Images/doc2524.pdf, [accessed 11-Dec-2012].

[8] E. Metz, R. Lencevicius, and T. F. Gonzalez, "Performance data collection using a hybrid approach," *SIGSOFT Softw. Eng. Notes*, vol. 30, no. 5, pp. 126–135, Sep. 2005.

[9] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: a call graph execution profiler," *SIGPLAN Not.*, vol. 39, no. 4, pp. 49–57, Apr. 2004.

[10] M. S. Müller, A. Knüpfer, M. Jurenz, M. Lieber, H. Brunst, H. Mix, and W. E. Nagel, "Developing scalable applications with vampir, vampirserver and vampirtrace." in *PARCO*, ser. Advances in Parallel Computing, C. H. Bischof, H. M. Bücker, P. Gibbon, G. R. Joubert, T. Lippert, B. Mohr, and F. J. Peters, Eds., vol. 15. IOS Press, 2007, pp. 637–644.

[11] M. Geimer, S. S. Shende, A. D. Malony, and F. Wolf, "A generic and configurable source-code instrumentation component," in *Proceedings of the 9th International Conference on Computational Science*, ser. ICCS 2009. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 696–705.

[12] R. M. Stallman and the GCC Developer Community, "Using the gnu compiler collection," http://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc.pdf.

[13] C. Weyer, C. Renner, V. Turau, and H. Frey, "Tinyaid: Automated instrumentation and evaluation support for tinyos," in *Proceedings of the Second International Workshop on Sensor Network Engineering (IWSNE'09)*, Jun. 2009.

[14] Atmel Corporation, "Atmel AT45DB161D," http://www.atmel.com/Images/doc3500.pdf, 2010, [accessed 11-Dec-2012].

[15] Micron, "Micron M25P80 Serial Flash Embedded Memory," http://www.micron.com/~/media/Documents/Products/Data%20Sheet/NOR%20Flash/5981M25P80.ashx, 2011, [accessed 11-Dec-2012].

[16] Graphviz, "DOT language," http://www.graphviz.org/doc/info/lang.html, [accessed 11-Dec-2012].

[17] F. Büsching, U. Kulau, and L. Wolf, "Architecture and Evaluation of INGA - An Inexpensive Node for General Applications," in *Sensors, 2012 IEEE*. Taipei, Taiwan: IEEE, oct. 2012, pp. 842–845.

[18] G. von Zengen, F. Büsching, W.-B. Pöttner, and L. Wolf, "An Overview of μDTN: Unifying DTNs and WSNs," in *Proceedings of the 11th GI/ITG KuVS Fachgespräch "Drahtlose Sensornetze" (FGSN)*, Darmstadt, Germany, 9 2012.

[19] A. Dunkels, "Full tcp/ip for 8-bit architectures," in *Proceedings of the 1st international conference on Mobile systems, applications and services*, ser. MobiSys '03. New York, NY, USA: ACM, 2003, pp. 85–98.

[20] W.-B. Pöttner, F. Büsching, G. von Zengen, and L. Wolf, "Data elevators: Applying the bundle protocol in delay tolerant wireless sensor networks," in *The Ninth IEEE International Conference on Mobile Ad-hoc and Sensor Systems (IEEE MASS 2012)*, Las Vegas, Nevada, USA, Oct. 2012.