Testbed Assisted Control Flow Tracing for Wireless Embedded Systems

Roman Lim, Lothar Thiele
Computer Engineering and Networks Laboratory
ETH Zurich, Switzerland
{lim, thiele}@tik.ee.ethz.ch

Abstract

Observing wireless embedded systems is difficult because of resource constraints and tight interaction with the environment. In this paper, we develop a method that can reconstruct the entire control flow of a program based on recorded state changes (time and state) of I/O pins. We instrument program binaries by statically inserting instructions that alter the states of a set of I/O pins. During program execution, the overhead of added instructions must be kept as low as possible to preserve the original program behavior. We first adapt an existing software-only placement method to generate an unambiguous pattern on the I/O pins for every possible execution path. Then, we make use of recorded timestamps to further reduce the runtime overhead substantially. This timing information is extracted from the executable by means of an elaborate static analysis. An algorithm is presented that safely reduces the number of recorded events while still being able to uniquely determine the executed program path. Experiments on a testbed show that using time information reduces the runtime overhead by up to 38.3% for typical applications. The average runtime overhead is 19%.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Tracing

General Terms

Design, Experimentation, Measurement, Algorithms

Keywords

Control flow tracing, instrumentation, time analysis, embedded systems

1 Introduction

Networked wireless embedded systems, known as sensor networks, cyber-physical systems or the Internet of Things have been applied in various fields: environmental monitoring [11], personal health and medicine [24, 2], and surveillance [23], to name only a few. Typically, components of such systems are small, low-power nodes that are equipped with a wireless communication module. Although there have been many successful deployments of such systems over the last ten years, building them is still a difficult task. Reasons for this are: (*i*) the distributed characteristics and unreliable communication channels of wireless embedded systems makes it difficult to build precise models. (*ii*) Nodes have a scarce energy budget, since batteries are heavy and expensive, and long periods of unattended lifetime are a prerequisite. Keeping costs and energy requirements low leads to hardware platforms that offer just enough memory and compute power for the task at hand [16]. Due to the limited resources, systems regularly operate on the limits of the available computing, energy and communication capabilities. Therefore, the risk of misbehavior in terms of functional and non-functional properties is high. Besides careful design approaches, it appears that extensive testing and debugging is essential for a successful design strategy [40, 25]. However, methods to increase observability and controllability of executed programs have to cope with very little resources.

Methods applied for debugging software on a single node range from simple LED observations, over `printf` statements to in-system debuggers. Testbed infrastructure extends the observability of program execution to an entire network. All these debugging methods provide insights into particular parts of the running program, but lack the ability to accurately trace the entire control flow of a program. As such, a program has to be (re-)instrumented every time for a specific goal. This is even true for in-system debuggers, since debugging interface bandwidths limit the extractable runtime information [29, 31].

The use of program flow tracing is not limited to debugging and failure diagnosis, but it is also applicable to program optimization or to collect software metrics like coverage [35, 26]. In the area of general purpose computing, software-only methods exist to completely trace program executions [20]. However, these methods have shown to be prohibitive when applied to wireless embedded systems [30]. Therefore, related approaches only trace a subset of the program, or instrument at a higher abstraction level, *e.g.*, function calls [19]. Some microcontrollers include dedicated program flow tracing hardware inside the chip. The embedded trace macrocell (ETM) in selected ARM chips allows to extract a data stream of executed instructions. However,

on-chip hardware debugging functionality comes at an additional cost in terms of die size and pin count [15]. The majority of recent node platforms, exemplified in Table 1, does not support hardware assisted tracing. In the real-time domain, approaches exist to measure execution times for worst case execution time analysis by tracing GPIO lines [9]. These solutions typically target more powerful processors, *e.g.*, MIPS or PowerPC.

Table 1. Recent node platforms. The Cortex-M ETM module allows to extract program flow traces.

Name	Year	Architecture	HW tracing
OpenMote [37]	2015	Cortex-M3	-
panStamp NRG 2 [1]	2015	MSP430	-
WandStem [33]	2016	Cortex-M3	ETM
OpenMote+ [36]	2016	Cortex-M4	ETM
Storm [3]	2016	Cortex-M4	-

Contributions and road-map. In this paper, we propose a novel hardware/software program flow tracing method that can be applied to trace the full program execution on instruction level in a pre-deployment testbed environment. Similar as [9], our approach relies on an external monitoring device capable of observing GPIO state changes, *e.g.*, a logic analyzer. Testbeds with similar monitoring capabilities [10, 21, 17, 27] allow to apply our approach to a large number of nodes. In contrast to architecture specific hardware debugging facilities, our approach only requires some spare GPIO pins, which makes it applicable to virtually any node platform.

Inserting program statements for the purpose of tracing adds runtime overhead, *i.e.*, additional CPU cycles. This overhead influences the program behavior and should therefore be minimal. We use information about execution time to reduce the runtime overhead of existing instrumentation approaches substantially. The execution time on the targeted systems' processor architectures is predictable due to the absence of caches, floating point units or branch speculation. We leverage this by extracting timing information from the executable by means of an elaborate static analysis.

We present an algorithm that reduces the number of recorded events while still being able to uniquely determine the executed program path.

In summary, this paper makes following contributions:

1. We design a method to trace program flow down to the instruction level using GPIO pin recording.

2. In Sec. 3, we present a new algorithm to reduce the number of emitted GPIO changes for tracing by exploiting time information.

3. In Sec. 4, we apply this algorithm to build a tool for MSP430 based platforms. It performs a static analysis of the program binary, adds instrumentation code and reconstructs traces from recorded GPIO events.

We experimentally show the influence of our new approach on different TinyOS and ContikiOS applications in a testbed of 31 nodes. The evaluation in Sec. 5 shows that our method adds an average runtime overhead of 19 %. The use of time information reduces the runtime overhead by up to 38.3 %.

In addition, we find that instrumentation has negligible influence on the reliability of Glossy [14], a timing sensitive flooding architecture that relies on constructive radio interference. In Sec. 6, we exemplify the usefulness of control flow tracing in two case studies.

2 Related Work

Tracing Program Execution. The problem of efficiently tracing and profiling program executions has been studied for several decades in the area of general purpose computing. One aspect of this problem is the question of where to best put witnesses (instrumentation code) in order to faithfully reconstruct the program flow [5]. Succeeding work found an efficient encoding of consecutive witnesses in program paths [6] and in whole programs [20]. We build on findings in [5] and extend these methods to make use of time information available when tracing embedded systems with an external observer. Since we can rely on external processing, our focus is to reduce the impact on the target system rather than to efficiently encode and compress traces. The aforementioned techniques cannot be directly applied to wireless embedded systems because of the scarce resources available on these devices.

Tracing program flow by monitoring GPIO lines has been done in the area of real-time systems [9, 39]. In particular, the pWCET tool instruments source code and traces program flow using this method [7]. However, the aim of these approaches is to *measure* the execution time of a program. In contrast, our method is based on a timing model of the processor and *uses* execution time as a means to substantially reduce the induced tracing overhead. Traces of either method can be used for measurement based worst case execution time analysis [8].

Wireless embedded systems. Software solutions for tracing wireless embedded systems instrument program code with logging instructions and store the generated trace in flash memory or transmit it over radio or serial communication interfaces. In [30], instrumentation code is inserted at branch instructions and an efficient encoding is used to log control flow traces to flash memory. Similarly, in [38] the authors instrument each basic block of the program and use time information to compress the generated trace. In contrast to [38], our solution avoids putting witnesses on every single basic block, and we can also handle nested loops where the exact number of iterations is unknown at compile time.

Another instrumentation approach is pursued by Tardis [32], which rather logs non-deterministic program inputs. These inputs are then fed to a simulator when replaying the program execution. A combination of control flow and data tracing is employed by LibReplay [19]. By logging function call arguments, program execution at function level is logged for replay.

While software solutions can provide very accurate information about the state of a node, the resources required for processing and storing the traces render such an approach unsuitable to trace time sensitive behavior. Indeed, experiments with Tardis reveal that the CPU duty cycle of standard node applications can almost double when tracing is enabled [32]. In addition, software instrumentation poten-
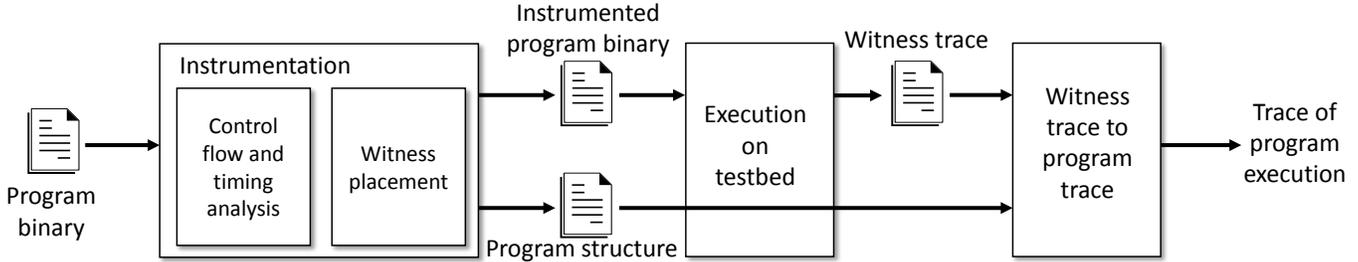
Figure 1. Overview of the tracing process.

tially produces data streams that easily exceed the data produced by the application itself, rendering instrumentation of the whole program prohibitive if the trace needs to be stored on the node itself.

Hardware Assisted Tracing. Additional hardware offloads data processing from the node to an external observer platform, providing an out-of-band communication channel. Aveksha [31] uses a debug board extension to trace events of interest on a single target node using the on-chip debug module of the MSP430 microcontroller. A low-cost and networked solution is provided by Minerva [29]. On-chip debug modules typically provide functions handy for interactive debugging (watch points, break points, reading and writing state information). Tracing the program flow by other means than polling the program counter is usually not possible. Reconstruction of the entire program flow might be possible if the polling interval is short enough. Recent processors include a special hardware tracing module to directly output trace information of the running program [4, 18]. However, typical sensor node platforms include lower end microcontrollers that do not include such features. Moreover, hardware debugging features are highly architecture dependent and therefore limited to specific node platforms.

Compared to existing hardware based tracing solutions, our approach is more portable and generally applicable since it only requires a few spare GPIO pins of a microcontroller instead of an architecture specific debug module.

3 Control Flow Tracing

An overview of our approach to control flow tracing is given in Fig. 1. We first statically instrument a program: the program binary is analyzed and witnesses are inserted at suitable locations. A witness is emitted whenever the control flow of the program passes its location. The instrumented binary is then loaded onto a set of nodes. During execution, we record the emitted witnesses. By combining the trace of witnesses with the extracted program structure, a trace of the program itself is reconstructed. In our case, witnesses are encoded into GPIO state changes and recorded using an external monitoring device.

Such an instrumentation has to fulfill the following requirements: (*i*) the emitted witness stream must be unambiguously mappable to the original program execution, and (*ii*) the runtime overhead of added instructions must be low in order to preserve the original behavior of the program.

Next, we first introduce in Sec. 3.1 the witness placement algorithm by Ball and Larus, which serves as baseline.

Then, in Sec. 3.2, we improve the baseline instrumentation approach by making use of timestamps taken by the recording device. Finally, in Sec. 3.3 we show how we efficiently encode witnesses using a limited number of GPIO pins.

3.1 Ball and Larus

The algorithm by Ball and Larus [5] efficiently places witnesses onto edges in a given control flow graph $G = (V, E, W)$. Every procedure (*e.g.*, function, interrupt handler) in a program is represented by a separate control flow graph. Basic blocks (groups of uninterrupted sequences of instructions) are associated with vertices $v \in V$, and directed edges $e \in E$ are transitions between basic blocks. Annotated edge weights $w \in W$ represent the expected number of times each transition is taken during program execution. Weights w can be obtained by profiling or by using a heuristic approach.

The algorithm optimizes the number of witnesses observed at runtime, *i.e.*, minimizing the total weight of all instrumented edges. As finding the minimal-cost solution is NP complete [5], Ball and Larus propose to use a heuristic to find a good solution: first, a maximal spanning tree on G is built. Then, witnesses are put on all edges not in the spanning tree. This procedure guarantees that there is only one possible witness-free path between any two witnesses.

To follow the program flow in between different control flow graphs, *blocking* witnesses are introduced. Blocking witnesses are placed in the control flow graph on edges preceding call sites or exits of functions.

3.2 Exploiting Time Information

Logic analyzers or testbeds with GPIO tracing abilities not only capture the states of the I/O pins, but also the time of the event, *i.e.*, *when* a state changed. Fig. 2 shows two
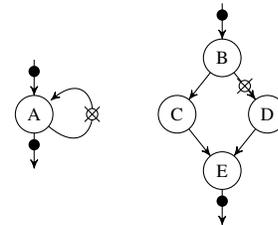

Figure 2. Execution time measurements allow to determine the control flow between two witnesses (black circles). Example for a simple loop (left), and for a graph with different execution times for each branch (right).
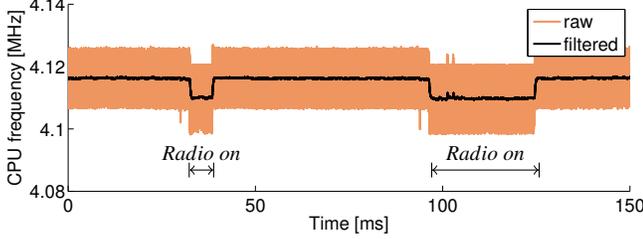
Figure 3. CPU clock speed measurement on an MSP430. The clock frequency varies, here up to 30 kHz (0.7 %).

excerpts of a control flow graph where time information can reduce the overhead of instrumentation. On the left, we can measure the execution time between the two witnesses (black circles) to infer the number of loop iterations. This renders the crossed out witness superfluous. On the right, if the execution times for the sequence $B - C - E$ is different from $B - D - E$, we can again infer the program flow from the time interval between the two remaining witnesses.

Both *time measurements* and *actual execution time* might be affected by uncertainties. Time measurements have a minimal time resolution, while execution times are affected by the stability of the processor's clock. Fig. 3 shows a measurement of the CPU's main clock on a TelosB node that was duty-cycling the radio transceiver. The resulting variation in power dissipation leads to changes in the supply voltage, and in consequence alters the speed of program execution. Based on these observations, we conclude that time uncertainties need to be well incorporated in a method that infers execution flow based on time measurements.

3.2.1 Problem Definition and Modeling

Program model. To describe the problem more formally, we extend the given control flow graph to $G = (V, E, W, T, B)$. As before, G is a directed graph, and every procedure of the program is represented by a single instance of G. Vertices $v \in V$ represent basic blocks and edges $e \in E$ transitions between them. Weights $w(e) \in W$ are the expected number of times a transition is taken at runtime.

In addition, execution times of basic blocks are annotated to vertices as costs $t(v) \in T$, and every execution of the procedure corresponds to a path in G, i.e. a sequence of vertices and edges. To take uncertainties in the execution times into account, T actually provides execution time intervals. The execution time $t(v)$ of the basic block associated to v is in the interval $t(v) \in T(v) = [l(v), u(v)]$. With each vertex v, there is also associated a bound $b(v) \in B$ which bounds the number of times the program flow may pass the corresponding basic block, possibly infinity if there is no bound known. In other words, any (feasible) path has $b(v)$ or fewer occurrences of vertex v in its sequence.

We add two special vertices to G, an *ENTRY* vertex that has an edge e_{entry} to the entry of the procedure, and an *EXIT* vertex that has incoming edges $e_{\text{exit},i}$ from every returning vertex. These elements are added for the sake of modeling and do not materialize in any real instrumentation code. A witness set $E_{\text{witt}} \subseteq E$ contains all edges that can be observed during execution. Paths are sequences of edges and vertices

in G. For the witnesses $e_i, e_j \in E_{\text{witt}}$, we denote the set of witness-free paths leading from e_i to e_j as $\text{path}^{\star}_{i \to j}$, i.e., all paths that can reach the edge e_j when starting at e_i without passing another edge of the witness set.

Problem definition. Our goal is to find a set of edges $e \in E_{\text{witt}} \subseteq E$ that minimizes the expected runtime overhead $C(E_{\text{witt}})$, i.e., the total weight of the edges in the set

$$C(E_{\text{witt}}) = \sum_{e \in E_{\text{witt}}} w(e). \quad (1)$$

To ensure that we can safely reconstruct the program path taken, it is mandatory that all the witness-free paths between any pair of witnesses and *ENTRY* and *EXIT* have distinguishable execution times. Therefore, we have $\{e_{\text{entry}}, e_{\text{exit}}\} \subseteq E_{\text{witt}}$, and for every pair of $e_i, e_j \in E_{\text{witt}}$ and for every pair of disjoint witness-free paths $p, q \in \text{path}^{\star}_{i \to j}$ between e_i and e_j it must hold that

$$\max \left(\sum_{v \in p} l(v), \sum_{v \in q} l(v) \right) - \min \left(\sum_{v \in p} u(v), \sum_{v \in q} u(v) \right) > \delta. \quad (2)$$

This condition ensures that the total execution time of two different paths between witnesses differs by more than the measurement's time resolution δ. In other words, in an admissible witness set there are no two disjoint witness-free paths between any pair of witnesses whose execution times cannot be distinguished. In the following, we refer to paths that are not distinguishable in time as paths *coinciding* in time.

3.2.2 Approach

The problem has the same objective as the baseline approach in Sec. 3.1, namely to minimize the number of times a witness is encountered during runtime. However, the conditions to meet are more relaxed. We do not require to have only one single witness-free path between two witnesses. We even allow cycles in the path, as long as the resulting paths do not coincide in execution time. While this might help to reduce the runtime cost of instrumentation, it makes the problem computationally more difficult to solve because program loops and possibly overlapping cycles in the graph lead to an exponentially growing number of paths between two witnesses. In the example shown in Fig. 4, there are two possible paths that can be taken within one loop iteration, leading to 2^n different possible paths of n iterations.

Because of these difficulties, we aim at finding a good heuristic rather than an optimal solution. The goal of the heuristic is to reduce the complexity of the problem while still being able to reduce the sum of the edge weights in the witness set. In the following, we apply two strategies for complexity reduction: (i) we consider only a subset of all edges in G to be eligible to bear witnesses, and (ii) we find a graph property that helps to quickly discern when paths between two witnesses are unlikely to be distinguishable.

In the following, we derive a witness placement method with an upper bounded cost function $C(E_{\text{witt}}) \leq C_0$.

3.2.3 Heuristic Overview

An overview on the heuristic is given in Algorithm 1. The key idea is to initially start with a feasible witness set $E_{\text{in}} \in E$.
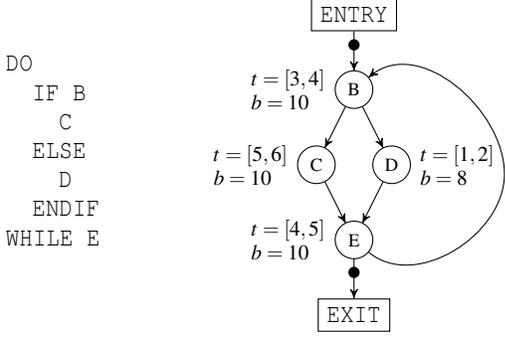
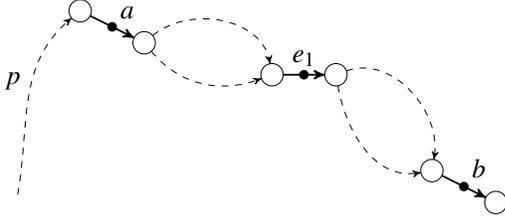**Figure 4. Example program with while loop (left) and resulting annotated graph (right); edge weights omitted.**

Figure 5. An initial witness set $\{a, b, e_1\}$ and witness-free paths (dashed lines). Neither removing a nor b can make e_1 removable, if e_1 cannot be removed with a and b.

Then, we successively try to remove every witness in the set, starting from the one with the largest weight $w(e)$. A witness can be removed if the remaining witness set does still fulfill (2), as verified with the function *isAdmissible* in Algorithm 1. Since we only remove edges from the set, the total weight of edges in the set can only decrease. The total weight of E_{in} is therefore at the same time a safe upper bound on the resulting cost of the heuristic. We select E_{in} to be the edges determined by the baseline approach without time information.

This approach has some favorable properties: (*i*) it guarantees that the resulting solution is at least as good as the baseline solution, *i.e.*, $C(E_{out}) \leq C(E_{in})$, (*ii*) it only needs to perform $|E_{in}|$ admission tests, which significantly reduces the search space, and (*iii*) for every admission test, only the subgraph that is affected by the removed witness has to be assessed, since we already start with a feasible witness set.

Algorithm 1 Instrumentation

Input: $G(V, E, W, T, B)$: Control Flow Graph, E_{in}: Initial feasible witness set
Output: E_{out}: Reduced witness set
1: $E_{out} \leftarrow E_{in}$
2: sort descending E_{out} according to $W(E_{out})$
3: **for all** $e \in E_{out}$ **do**
4:     **if** isAdmissible(G, $E_{out} \setminus e$) **then**
5:         $E_{out} \leftarrow E_{out} \setminus e$
6:     **end if**
7: **end for**

To see why only $|E_{in}|$ tests are necessary, let us consider the example in Fig. 5. Suppose that we first try to remove e_1 from the witness set. If we cannot remove this witness, it means that there must be at least two paths connecting two
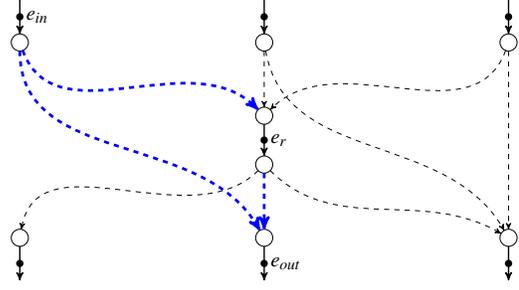
other witnesses through e_1 with coinciding execution times. Let us call these other witnesses a and b, *i.e.*, the coinciding paths start at a and end at b. e_1 cannot be removed as long as those witnesses exist. Suppose a is removed in a consecutive step of Algorithm 1 and e_1 revisited. All witness-free paths previously leading to a reach now to e_1. If we remove now e_1, a path p that had led to a is now extended with all possible paths leading from a to b. Since we know that at least two paths from a to b coincide in time, two of the resulting paths must coincide as well, and therefore we must not remove e_1. A similar argument can be made for when removing b. In summary, the admission test has to be done only once for each witness; a single iteration over all witnesses in the initial set E_{in} suffices.

3.2.4 Test for Distinguishable Paths in a Subgraph

Let us now focus on the *admission test*, which is performed by function *isAdmissible* in Algorithm 1. The purpose of this test is to check whether the remaining set of witnesses still fulfills the condition (2) on G.

Since we start with a feasible witness set, the test only needs to operate on the subgraph SG affected by the witness e_r to be removed, *i.e.*, including all vertices and edges that lay on a witness-free path between witnesses that can reach e_r, or that can be reached from e_r, as illustrated in Fig. 6. In this example, the dashed paths need to be examined, other parts of the graph are not affected by e_r and are therefore omitted in the figure. As the uniqueness needs to be shown for each affected witness pair, we first reduce (for convenience) the subgraph SG. To this end, we choose one pair of witness edges e_{in}, e_{out} and remove all edges and nodes that are not reachable from e_{in} and cannot reach e_{out}, *i.e.*, we keep only the bold dashed paths in Fig. 6.

For each choice of e_{in}, e_{out}, all possible witness-free paths must be distinguishable in execution times for a feasible solution. For paths that do not contain any program loops, checking (2) is straightforward: the upper and lower range of a path can be calculated by summing up the costs of all vertices in the path. However, if a path contains loops, the number of different possible execution times to assess grows exponentially, as discussed in Sec. 3.2.2.

Our approach to mitigate this effect is as follows: we derive a necessary condition that needs to be fulfilled for unique executions times of paths containing loops. Graph structures

that result from removing a witness and that do not fulfill this condition can be quickly rejected for admission.

Reasons for Non-unique Paths. For the later discussion, we distinguish two reasons for non-unique paths:

- Two paths have exactly the same number of occurrences of each vertex, just in a different order. As a result, the accumulated execution times on each path are equal and therefore, the two paths cannot be distinguished. We say that two paths with the same number of occurrences of each vertex are equivalent.

- Two paths have different numbers of occurrences of each vertex, but the difference in (2) is not larger than δ.

In the following, we derive a condition that is necessary to exclude the existence of equivalent paths.

Equivalent Paths. The (reduced) control flow graph SG can first be decomposed into strongly connected components (SCCs), *i.e.*, into maximal subgraphs where there is a path in each direction between each pair of vertices of the graph. This is exemplified in Fig. 7 (a). If each SCC is contracted to a single vertex, the resulting graph is a directed acyclic graph (see Fig. 7 (b)). In other words, the program flow follows the acyclic graph (never returns to previous subgraphs) but may do loops within each SCC.

The program path through the acyclic graph is unique in the number of occurrences of each vertex. Therefore, the only non-uniqueness can come from one of the SCCs.

Let us suppose an SCC contains a vertex v with at least two outgoing edges (v,i) and (v,j) where $b(i) \geq 1$ and $b(j) \geq 1$ is satisfied and with $b(v) > 1$. According to Sec. 3.2.1, $b(v)$ is an upper bound on the number of times v can be executed. Then it may be possible that v is visited twice in a program path: first, the program path follows edge (v,i) and afterwards it follows edge (v,j). Now, there may exist another program path where the order of visit is reversed, i.e. at first (v,j) and then (v,i). This can be seen as follows: The considered subgraph is an SCC. Therefore, there exists a path from every vertex to every other vertex and therefore, there exists paths that connected i as well as j back to vertex v.

The reverse is true as well: if there does not exists such a vertex v in any SCC, then there are no equivalent paths. This can be shown as follows: if there exist two equivalent paths, then there is a first vertex after which the two paths are different, but vertices are visited equally often. Let us call this vertex v and the two succeeding vertices i and j. As both edges can be taken, we have $b(i) \geq 1$ and $b(j) \geq 1$. Clearly, the node v needs to be visited at least twice as the both vertices i and j are part of both paths (they need to be visited equally often). Therefore, $b(v) > 1$ is necessary.

In summary, if an SCC contains a branch v, then under relatively general conditions ($b(i) \geq 1$, $b(j) \geq 1$, $b(v) > 1$) we cannot exclude the existence of equivalent paths. In other words, a safe and not too restrictive assumption is that SCCs of the program graph should contain a single cycle only, without any internal branches. We use this finding in our heuristic to early reject the removal of a witness. Strongly connected components in a graph can be found in linear time [13].

Path Differences. Now we can address the problem of comparing path lengths in SG according to (2). As discussed in the previous section, we dismiss all subgraphs that contain SCCs with branches, *i.e.*, the remaining subgraph contains only SCCs with single loops. For execution time calculation, we again use the decomposition of the reduced control flow graph into SCCs and the contraction of each component into a single vertex.

In terms of accumulated execution times, we have now a new control flow graph, *e.g.*, as shown in Fig. 7 (c), with the following properties:

- The new control flow graph has (as before) initial and final edges e_{in} and e_{out}.

- Each SCC is replaced by an acyclic subgraph with as many input and output edges as the product of the number of original inputs and outputs to the SCC. The acyclic subgraph has no internal edges. The vertex cost between one pair of input and output edges has two parts: The additive part $t_0(v)$ equals the sum of vertex costs (execution times) of the original program if the program path does not contain a cycle; the multiplicative part $t_c(v)$ equals the sum of execution times of a complete cycle execution. In other words, we have

$$t(v) = t_0(v) + n(v) \cdot t_c(v)$$

The factor $n(v)$ determines the number of complete cycle executions. This number can be bounded from the corresponding bounds of the original control flow graph.

- These acyclic subgraphs are connected by the rest of the control flow graph, i.e. all vertices that do not belong to a SCC. The vertex costs correspond to the original execution times of the control flow graph.

In summary, we now have an acyclic control flow graph $G = (V, E, W, T_0, T_c, B)$ where each vertex has a cost of the form $t(v) = t_0(v) + n(v) \cdot t_c(v)$ where $n(v) \leq b(v)$.

Now, testing (2) can be done as follows: the acyclic control flow graph is traversed in topological order. During the

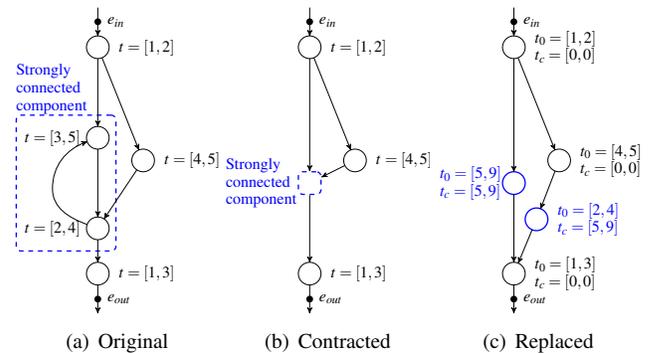(a) Original   (b) Contracted   (c) Replaced

Figure 7. Example graph with one strongly connected component (SCC). Contracting the SCC in (a) leads to (b). In (c), the SCC is replaced by an acyclic subgraph, consisting of new vertices with additive and a multiplicative execution times t_0 and t_c.

traversal, sets of intervals are maintained and updated. At join nodes, two sets are joined and if two intervals overlap, paths cannot be distinguished and the traversal can stop. If a set passes a vertex v, then we replace the set by a new one that contains for each interval I in the original set the intervals $I + t_0(v) + n(v) \cdot t_c(v)$ for all $n(v) \leq b(v)$. If an added interval overlaps with an existing one in the new set, the traversal stops as paths cannot be distinguished anymore.

Finally, we test whether the intervals in the resulting set (at e_{out}) are separated by at least δ, i.e., the distance between the upper bound of any interval and the lower bound of the subsequent interval is at least δ. If no paths are coinciding in execution time for any pair of e_{in}, e_{out} in SG, removing that particular witness from the initial witness set is admissible. We show in Sec. 5.1 that the off-line computational complexity of the admission test is small for realistic programs.

3.3 Encoding of Witnesses

To instrument a program, we have to define an encoding that maps witness identifiers to GPIO state changes. We assume to have N pins that we can observe. The number of witnesses $|I|$ that can occur is $|E| - |V| + 1$ in the case of Ball and Larus, i.e., there is a witness on every edge in the control flow graph G but on those that are part of the spanning tree. In the case where $2^N > |I|$ each witness can be binary encoded with at most one state change per pin. Depending on the processor architecture, this can be managed by a single instruction if all pins belong to the same port on the microcontroller, e.g., using XOR <IDValue>, <GPIO port>. For more witnesses, we need an encoding that can represent witnesses using sequences of GPIO state changes with a minimal amount of CPU cycles. We employ two strategies to achieve this goal: (*i*) reuse witness identifiers, and (*ii*) encode identifiers that are used more often using cheaper codes.

Reusing Witness Identifiers. Witnesses in different control flow graphs can have the same identifiers, since the blocking witnesses allow us to unambiguously determine transitions between control flow graphs. Within a control flow graph, for every witness, every set of reachable witnesses must have unique identifiers. This problem can be modeled as finding a proper vertex coloring in an undirected graph where each witness is a vertex. For every set of reachable witnesses, we add edges between all pairs of witnesses in the set. Finding a minimal number of unique identifiers is equal to finding a minimal number of colors for the vertex coloring problem.

Identifier Encoding. For the remainder, we assume a hardware platform that supports to set or change the state of all monitored pins at once. One state change is the smallest unit in terms of cost, and allows to represent 2^N different codes. Given a set of identifiers $m \in M$ and an expected occurrence probability $p(m)$, our goal is to find an encoding $C(m)$ that minimizes the total cost $R = \sum_m p(m) \cdot \text{len}(C(m))$. To minimize this cost, we use k-ary Huffman coding [28] with $k = 2^N$.

Nested Witnesses due to Interrupts. Witnesses might be encoded using more than one single GPIO state change. When an interrupt occurs, witnesses emitted within that interrupt handler might separate an ongoing witness in the interrupted program and therefore alter the observed GPIO

code of that witness. To avoid such ambiguities, we instrument every start of an interrupt handler with a globally unique GPIO code. Whenever a unique code is observed during trace reconstruction, the parts of the trace belonging to an interrupt can be safely separated from other parts.

4 Implementation

In this section, we present a tool that implements our instrumentation method for MSP430 based platforms. The MSP430 series is a family of low-power microcontrollers, featuring a 16-bit RISC CPU [34]. GPIO pins, organized in ports of 8 bits, are accessed using memory mapped I/O registers. The main clock of the CPU is sourced either by an external quartz oscillator or by an internal digitally controlled oscillator (DCO) with RC-type characteristic. The MSP430 architecture has no caches and no branch speculation, which makes execution timing deterministic. Each instruction takes a fixed number of CPU cycles to complete, depending on the addressing mode of the instruction. Sleep states disable one or several peripherals and clocks to save energy. Any sleep state disables the CPU clock.

We implement instrumentation and replay function in a Python based tool, which we also make available for download[1]. Replay is the process of reconstructing the program execution based on the recorded events. In the following, we describe the two functionalities in more detail, and we discuss selected design decisions.

4.1 Binary Instrumentation

The instrumentation process takes a program binary as input and automatically adds the necessary instrumentation code. Instrumentation works on the level of machine code instructions because our approach needs accurate information about execution timing. Instrumenting programs in high level programming languages is not an option, since the exact realization and timing highly depend on the compiler and the optimizations performed during compilation.

There are several steps involved in program instrumentation: (*i*) the program structure is analyzed and a control flow graph for each function or procedure is extracted. (*ii*) We then apply our heuristic algorithm to select edges that need to be instrumented with a witness. (*iii*) Instructions that change GPIO states are inserted into the program.

Program flow analysis. This step extracts the control flow graph for every function or procedure in the program. Each instruction of the executable is parsed and grouped into basic blocks, i.e., instructions that form a continuous program flow. To connect basic blocks in the graph, all possible successors of the block's last instruction need to be known. A jump instruction or a conditional branch directly contains the target address. More complicated are indirect branches, which often result from C-switch statements. Indirect branch instructions operate on addresses stored in registers, and therefore the calculation of the address needs to be understood for a general solution. However, we find that a template based approach works well if targeting only one specific compiler. A template is a set of rules that is applied to instructions preceding the indirect branch instruction to find the base address and the size of the jump table involved.

---

[1] https://github.com/rolim/msp430-tracing

Interrupts and sleep states. In low-power applications, microcontrollers commonly spend most of the time in a sleep state to preserve power. To properly trace the program execution, we have to keep track of the power state of the processor. Sleep states are exited by interrupts. On the other hand, interrupts might also occur while the processor is active. On the MSP430, the current low-power mode is configured by flags of the status register. Before interrupt handler execution, the status register is pushed onto the stack. Thus, to track the low-power states of the processor, we insert additional witnesses at places where the program might enter or exit a low-power mode, *i.e.*, at the start of an interrupt handler, or at instructions that modify the low-power mode flags of the status register. A witness at the beginning of an interrupt handler also encodes the last sleep state as stored in the status register on the stack.

Instrumentation code. Instructions that represent a witness should be cheap, both in terms of execution time and size. On the other hand, they must be correctly decodable, irrespective of the program flow and the current state of the GPIO pins. We considered two different possibilities to fulfill these requirements: (*i*) Every witness translates into a single *exclusive or* operation on the identifier and the current port state. (*ii*) Every witness consists of two instructions, one that resets the state of the GPIO port, and one that sets the value to the actual identifier of the witness. The first option requires to restore the status register of the CPU in case a succeeding instruction is influenced by a status flag (*zero*, *negative*, ...) because the *exclusive or* affects the status register. The second option has no functional impact on other instructions. We decided to use the first option, since it has lower overhead, and since we encountered the need for restoring the status register only very rarely.

Effects of instrumentation. Inserting additional instructions into an existing program might change addresses of existing instructions. Therefore, instructions relating to such addresses (*e.g.*, function calls, branches, ...) need to be adapted. Some instructions might need to be replaced, *e.g.*, relative jump instructions on the MSP430. These instructions can perform jumps to addresses located within 512 bytes of the jump instruction. Added instrumentation code might now lead to jumps that exceed this limit, and therefore require to replace a relative jump with an absolute jump.

Changes in the program alter the execution timing of the program, and therefore also change the conditions that did hold during the placement of the witnesses. We resolve this issue by iterating the instrumentation process as long as there are no more changes needed. In every iteration, we update the control flow graphs to reflect the altered timing behavior. We then re-run the witness placement heuristic on the updated control flow graphs. Now, this could lead to an oscillation or even an infinite loop, *e.g.*, if placing an instruction makes and ambiguous path distinct again. To prevent this from happening, we only allow each iteration to add additional witnesses but not remove old ones.

4.2 Replay

During replay, we translate recorded GPIO events, *i.e.*, tuples (time, pinnumber, pinstate), to program addresses. First,

we group events with the same time. Then, we lookup the resulting witness identifier. By following the path to the next recorded witness in the control flow graph, we can gradually reconstruct the control flow during execution. In case there are multiple possible paths between the current location and the next witness, we choose the path that has an execution time closest to the measured execution time. Our heuristic ensures that this is sufficient to remove path ambiguities.

5 Evaluation

We evaluate the impact of our tracing method on five different applications. All experiments run on FlockLab [21], a public testbed with 31 nodes, which provides a service to trace GPIO states. We assess the impact of GPIO tracing with three increasing levels of information: (*i*) witness ID only, (*ii*) with timestamps of witnesses, and (*iii*) including upper bounds of loops. For every setting, we investigate the impact on program size and runtime overhead. Our experiments reveal the following key findings:

- Instrumentation with knowledge of upper bounds adds the lowest *static memory overhead* to the program binary, on average 31 % (6.8 kB). Compared to the witness-only case, upper bounds can reduce the number of non-blocking witnesses by 67 %.

- The *runtime overhead* in terms of CPU cycles to trace the entire program flow is between 14 % and 21 % (19 % average) using upper bounds. This is 13.5 % and 38.3 % less than with the baseline approach.

- Programs performed similarly with and without instrumentation. Glossy, which relies on exact timing of actions in order to generate constructive radio interference within the network, achieved at least 99.978 % data yield with any of the three instrumentation types.

5.1 Experimental Setup

To show the feasibility and the potential of GPIO tracing, we selected five example applications that are optimized for different target scenarios. Apart from the TinyOS *Blink* application, all candidates form a multi-hop network. *Multihop Oscilloscope (MHO)*, available from the TinyOS repository, is a general purpose data gathering application. It samples a sensor value every second and reports these values every five seconds to a base station. We choose to run this application with and without the low-power listening MAC protocol (LPL). In the LPL configuration, we set the wakeup interval to 512 ms. *Dozer* [12] is optimized for very low duty cycles and low data rates, generating a data packet on each node every 30 seconds. Finally, we include *Glossy* [14], which provides a fast and energy efficient flooding architecture. We use a flooding period of 2 seconds. This selection of example applications also covers two popular sensor network operating systems, namely TinyOS and ContikiOS. To illustrate the complexity level of each program, we list the program size and number of basic elements (functions, basic blocks and instructions) of each binary compiled for the TelosB platform in Table 2.

We instrument each application with three different levels of information to assess the benefits of additional information with respect to memory overhead and runtime over-

Table 2. Binary size and number of program elements for applications used in the evaluation.

Application name	Operating system	Size	Functions	Basic blocks	Instructions
Blink	TinyOS	2564	19	259	875
Multihop Oscilloscope	TinyOS	24522	175	2728	8329
Multihop Oscilloscope LPL	TinyOS	26128	188	2967	8848
Dozer	TinyOS	33798	203	3347	11090
Glossy	ContikiOS	16128	125	1556	5395

head. First, we rely only on the recorded witness identifiers. Since this approach does not use any time information, there must be only one possible program path between two consecutive witnesses. This approach is the most expensive in terms of overhead. Then we make use of witness timestamps. As explained in Sec. 3.2, by taking execution times into account, we can remove witnesses from the previous approach, as long as paths between consecutive witnesses have distinguishable path lengths. Based on empirical measurements, we assume a time inaccuracy of 1%. We assume to have no information about loop upper bounds (*i.e.*, the upper bound is infinity), and thus this approach cannot remove witnesses within loops. The two instrumentation methods so far do not require any specific runtime information about the program, and can therefore be applied without any further information about runtime execution. To further reduce the number of witnesses and therefore the runtime overhead of our tracing method, we add information about upper bounds of loops. For our experiments, we use a measurement based approach do get upper bounds. More specifically, we profile each application and count the maximal number of iterations of each loop. We then add a margin of 20% and use this value as upper bound. In the following, we refer to these three different variants as *w/o time*, *time* and *time & bounds*. The instrumentation process for any of the 15 binaries took less than 2 minutes each on a standard PC. As instrumentation adds witnesses at the beginning of interrupt handlers (see Sec. 4.1), we adapt the time compensation code of Glossy in one handler by adding a constant value to the timer capture register[2].

5.2 Static Overhead

The number of added witnesses for each instrumented application is shown in Table 3. A key figure is the number of non-blocking witnesses that remain to be instrumented. As described in Sec. 3, blocking witnesses are required to retrace the program flow between functions. Since we need this property in any of the three instrumentation variants, the timing aware approaches can only remove non-blocking witnesses. Fig. 8 shows that time analysis cuts the number of non-blocking witnesses in half. Using loop upper bounds further reduces the number of witnesses. In the case of Multihop Oscilloscope LPL, we see a total reduction in non-blocking witnesses of 67%.

The static overhead in terms of program memory is of practical importance, since program memory is a scarce resource in wireless embedded systems. In Fig. 9, we compare

---

[2]This modification can be done in a semi-automatic manner by providing placeholder variables that are filled with the number of added CPU cycles during the instrumentation process.
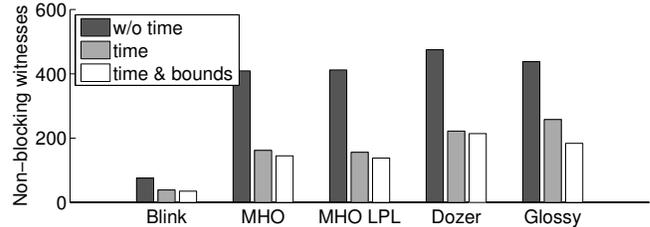

Figure 8. Number of non-blocking witnesses in each instrumented binary (last three columns of Table 3).

the size of the instrumented binary to the original one. We find that the memory overhead ranges from 23% to 49%. As expected, having timestamps and upper bounds each reduces the size of the instrumented binary. The most efficient variant (*time & bounds*) adds a memory overhead of at most 38% for all the investigated examples. While this is not a negligible amount of memory, it still allows us to fully instrument programs that use up to 72% of their total available flash memory, *e.g.*, 34.7 kB on a TelosB node. Our tracing method does not require additional RAM.
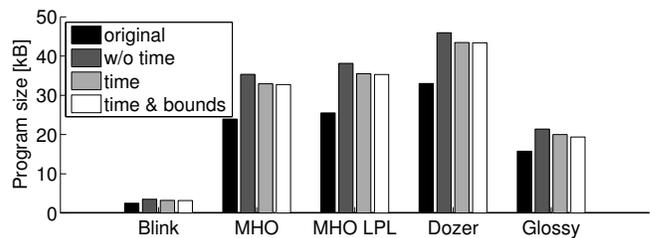

Figure 9. Memory utilization for original program and instrumented variants. No additional RAM is needed.

5.3 Runtime Overhead

We use FlockLab's GPIO tracing service [22, 21] to record the states of the GPIOs during execution. FlockLab allows to trace up to 5 GPIO pins on every node. We let every combination of application and instrumentation type run for 30 minutes on 31 TelosB nodes. Then, we replay all the traces and count the number of CPU cycles spent on emitting witnesses and the CPU cycles used for original instructions.

The runtime overhead is shown in Fig. 10. The average overhead per experiment ranges from 14% to 28%. Similarly as for the static overhead, the instrumentation method using loop upper bounds adds the lowest overhead (14% to 21%). The advantages of the upper bound approach are more pronounced because removing witnesses in a loop has a bigger impact on runtime overhead than on memory overhead. We also find that the runtime overhead (avg. 19%) is smaller

Table 3. Number of blocking and non-blocking witnesses used for instrumentation.

Application name	Blocking witnesses			Non-blocking witnesses		
	w/o time	time	time & bounds	w/o time	time	time & bounds
Blink	70	70	70	76	39	35
Multihop Oscilloscope	916	916	916	409	162	145
Multihop Oscilloscope LPL	1022	1022	1022	412	156	138
Dozer	1113	1113	1113	475	222	214
Glossy	390	390	390	438	258	184

than the program memory overhead (avg. 31 %) would suggest. This is mainly due to instrumentation instructions on the MSP430 being larger than the average, but execution wise around average. Comparing *w/o time* to *time & bounds*, we see reductions between 13.5 % and 38.3 %.

Since Glossy requires tight synchronization between concurrent transmitters in order to achieve constructive interference [14], we verified the performance of all three Glossy runs by comparing the performance metrics *reliability* and *radio duty-cycle* to the unmodified program. As shown in Table 4, all three test runs exhibit a high reliability of at least 99.978 % and low radio duty-cycle. In addition, there are only slight variations between tests.
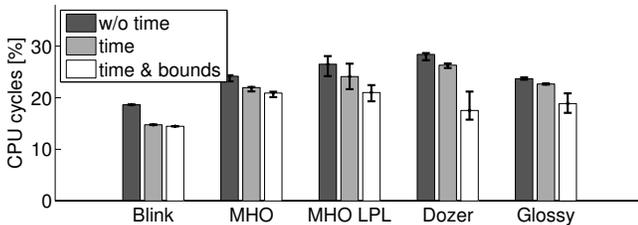
Figure 10. Runtime overhead caused by added instructions. Error bars indicate minimum and maximum.

Table 4. Reliability and radio duty cycle of Glossy.

Instrumentation type	Reliability	Radio duty-cycle
Unmodified	99.996 %	0.57 %
w/o time	99.986 %	0.60 %
Time	99.996 %	0.59 %
Time & bounds	99.978 %	0.59 %

These experiments show that full control flow tracing in a testbed is feasible and adds a relatively low runtime overhead. Moreover, it is also suitable for tracing time sensitive applications.

6 Case Study

In the following, we give two examples that show how program traces can be leveraged to analyze program behavior in wireless embedded systems. We extract and analyze code coverage, and we use traces to inspect timing behavior.

6.1 Code Coverage

We examine code coverage statistics that can be directly extracted from execution traces. Code coverage describes the amount of code that has been executed during runtime. It is used to measure the amount of program code that is covered by a set of tests. Test suites generally aim at maximizing

their code coverage [26]. We calculate the code coverage as the percentage of distinct executed basic blocks during a program run.

Nodes with similar code coverage might take on similar roles in a network, *e.g.*, acting as sink or leaf node, or relaying messages. Examining the diversity of code coverage might help to understand these systems better, and to tailor test cases to specific scenarios.

In the following, we analyze the traces obtained in Sec. 5 and calculate the overall code coverage and the distance of code coverage between nodes. We define the distance D between two sets of covered basic blocks Γ_i, Γ_j as the Hamming distance between the sets, *i.e.*,

$$D(\Gamma_i, \Gamma_j) = |\Gamma_i \cup \Gamma_j| - |\Gamma_i \cap \Gamma_j|.$$

Overall, we find that the test runs cover 62 % to 67 % of all basic blocks in the program. Specific numbers are given in Table 5. Interestingly, even a simple program like Blink does only cover two third of the entire program. Most of the uncovered parts of Blink are related to interrupt handlers (Timer A) that are never used during program execution, but still included in the binary. Pointers to uncovered program parts can help the developer to eliminate unused code in order to save on scarce program memory.

Table 5. Code coverage of example applications.

Application name	Code coverage	Basic blocks
Blink	66 %	171
Multihop Oscilloscope	65 %	1761
Multihop Oscilloscope LPL	67 %	1997
Dozer	62 %	2079
Glossy	65 %	1014

To analyze the variability in code coverage across the network, we calculate two coverage metrics from the traces: the average distance between sink node and the rest of the network, and the average pairwise distance between all nodes but the sink node. These metrics are shown in Fig. 11. Generally, a sink node executes clearly a different set of basic blocks than the rest of the network. For Blink, all nodes have exactly the same code coverage, as would be expected, since the program runs independently on every single node. In the case of multi-hop applications, there seem to be two distinct cases that differ to each other by the amount of variation between the non-sink nodes. Applications running on top of topology based network protocols (MHO, MHO LPL, Dozer) exhibit larger variations than approaches based on network flooding (Glossy). In the latter case, coverage dif-

fers only within 24 basic blocks among all non-sink nodes. This can be explained as follows: in topology based approaches, the executed code paths depend on the node's position in the topology, *e.g.*, compared to a node close to the sink, a leaf node does not need to forward any messages. In Glossy, all nodes participate similarly in every flood, irrespective of their position in the network.
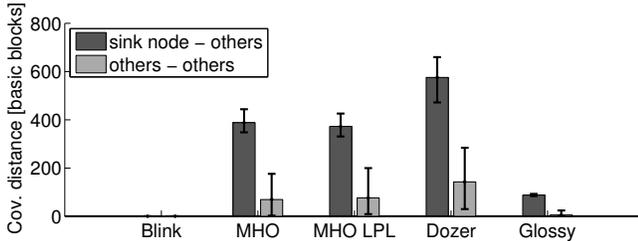

Figure 11. Average pairwise distance of code coverage between nodes in the experiment. Error bars indicate minimum and maximum.

Control flow tracing allows to extract useful aggregated information from program executions. Code coverage can be used to reason about program behavior in a network, or to find potentially unnecessary code, thus helping to reduce program size.

6.2 Inspecting Time Behavior

Erroneous or unwanted timing in embedded systems can lead to malfunction or degraded performance (*e.g.*, higher energy usage, or lower throughput). Runtime traces can give pointers to problematic parts of a program.

In this section, we analyze the timing within a part of the boot sequence on a TelosB node. During our experiments for the evaluation of this paper we realized that boot time can vary by several seconds among nodes when running ContikiOS. Traces show that most of the time is used for calibrating the digitally controlled oscillator (DCO). This DCO has to be configured by software to run at the desired target frequency, 4 MHz in our case. The actual frequency of the DCO is measured using the low frequency crystal oscillator on the TelosB as reference.

Fig. 12 shows the actual DCO frequency during the calibration process on TinyOS and ContikiOS. TinyOS uses a binary search, achieving calibration within tens of milliseconds, while ContikiOS performs a linear sweep (steps are caused by overlapping frequency ranges within the sweep). A code excerpt of the latter is provided in Listing 1[3]. Apart from the longer convergence time during the sweep, the calibration routine of ContikiOS exhibits a suspicious tail after reaching the target frequency. This tail turns out to be the main source of boot time variability among nodes. By inspecting the traces, we find that this period of almost constant frequency is caused by the calibration routine oscillating around the target frequency. According to Listing 1, line 103, the calibration ends if the measured number of clock cycles *exactly* matches DELTA, *i.e.*, the number of DCO cycles during one reference clock cycle at the target frequency. Due to the granularity of frequency control, it might take a

---

long time until this condition is met, even though the actual frequency cannot be tuned closer to the target frequency. To remove this random behavior, we suggest to add a test for oscillation, or to replace the calibration routine with a similar approach as in TinyOS.
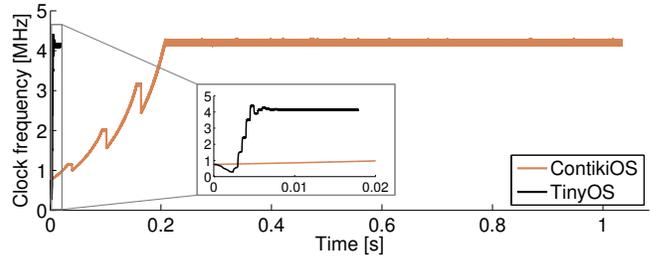

Figure 12. CPU speed profile at boot time for two different clock calibration algorithms.

Listing 1. Initial DCO calibration in ContikiOS.

```
95   while(1) {
96
97     while((CCTL2 & CCIFG) != CCIFG);
98     CCTL2 &= ~CCIFG;
99     compare = CCR2;
100    compare = compare - oldcapture;
101    oldcapture = CCR2;
102
103    if(DELTA == compare) {
104      break;
105    } else if(DELTA < compare) {
106      DCOCTL--;
107      if(DCOCTL == 0xFF) {
108        BCSCTL1--;
109      }
110    } else {
111      DCOCTL++;
112      if(DCOCTL == 0x00) {
113        BCSCTL1++;
114      }
115
116    }
117  }
```

This example shows that the increased observability provided by control flow traces can greatly facilitate the analysis of embedded systems. Time behavior can be analyzed in detail without tailoring an instrumentation strategy to a specific goal, *e.g.*, using counter registers and printf statements.

7 Conclusions and Future Work

We have presented a novel testbed based method that allows to completely trace the program flow of wireless embedded systems. By inserting instructions that encode witnesses of the program flow using GPIO state changes, the execution of a program can be observed down to instruction level. To this end, we designed a new algorithm that uses time information to reduce runtime overhead of instrumentation substantially, while still being able to uniquely determine the executed program path. Our experimental evaluation showed that our approach has an average runtime overhead of 19%. Compared to an approach without time analysis, using time information reduces the runtime overhead by up to 38.3%. This makes our approach also suitable to trace timing sensitive applications.

By enabling program tracing on a wide range of wireless embedded systems, we provide a tool that can serve as the

starting point for new debugging methods, automated program verification or optimization.

Although we implemented tracing for the MSP430 architecture, our approach is more general, and we want to see it also ported to other architectures in the future. We also see the need for tools that facilitate browsing and inspection of traces, especially for traces recorded from distributed systems and networks.

Acknowledgements

8 References

[1] Panstamp NRG 2. http://panstamp.com/.

[2] H. Alemdar and C. Ersoy. Wireless sensor networks for healthcare: A survey. *Comput. Netw.*, 54(15):2688–2710, Oct. 2010.

[3] M. P. Andersen, G. Fierro, and D. E. Culler. System design for a synergistic, low power mote/BLE embedded platform. In *Proc. of the 15th Intl. Conf. on Information Processing in Sensor Networks (IPSN)*, 2016.

[4] ARM. Embedded trace macrocell, architecture specification, Sep. 2011.

[5] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, July 1994.

[6] T. Ball and J. R. Larus. Efficient path profiling. In *Proc. of the 29th Annual ACM/IEEE Intl. Symp. on Microarchitecture (MICRO)*, 1996.

[7] G. Bernat, A. Colin, and S. Petters. pWCET: a tool for probabilistic worst-case execution time analysis of real-time systems. Technical Report YCS-2003-353, University of York, 2003.

[8] G. Bernat, A. Colin, and S. M. Petters. WCET analysis of probabilistic hard real-time systems. In *Proc. of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, 2002.

[9] A. Betts, N. Merriam, and G. Bernat. Hybrid measurement-based WCET analysis at the source level using object-level traces. In *Proc. of the 10th Intl. Workshop on Worst-Case Execution Time Analysis (WCET)*, 2010.

[10] S. Bouckaert, W. Vandenberghe, B. Jooris, I. Moerman, and P. Demeester. The w-iLab.t testbed. In *Proc. of the 6th ICST Intl. Conf. on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom)*, 2010.

[11] B. Buchli, F. Sutton, and J. Beutel. GPS-equipped wireless sensor network node for high-accuracy positioning applications. In *Proc. of the 9th Europ. Conf. on Conference on Wireless Sensor Networks (EWSN)*, 2012.

[12] N. Burri, P. von Rickenbach, and R. Wattenhofer. Dozer: Ultra-low power data gathering in sensor networks. In *Proc. of the 6th Intl. Conf. on Information Processing in Sensor Networks (IPSN)*, 2007.

[13] E. W. Dijkstra. *A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 1997.

[14] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh. Efficient network flooding and time synchronization with Glossy. In *Proc. of the 10th Intl. Conf. on Information Processing in Sensor Networks (IPSN)*, 2011.

[15] S. B. Furber. *ARM system-on-chip architecture*. pearson Education, 2000.

[16] M. Healy, T. Newe, and E. Lewis. Wireless sensor node hardware: A review. In *Proc. of the 7th IEEE Conf. on Sensors*, 2008.

[17] W. Huangfu, L. Sun, and J. Liu. A high-accuracy nonintrusive networking testbed for wireless sensor networks. *EURASIP Journal on Wireless Communications and Networking*, 2010(1):1, 2010.

[18] Intel. Architecture, instruction set extensions programming, reference, Dec. 2013.

[19] O. Landsiedel, E. M. Schiller, and S. Tomaselli. LibReplay: Deterministic replay for bug hunting in sensor networks. In *Proc. of the 12th Europ. Conf. on Wireless Sensor Networks (EWSN)*, 2015.

[20] J. R. Larus. Whole program paths. *SIGPLAN Not.*, 34(5):259–269, May 1999.

[21] R. Lim, F. Ferrari, M. Zimmerling, C. Walser, P. Sommer, and J. Beutel. FlockLab: A testbed for distributed, synchronized tracing and profiling of wireless embedded systems. In *Proc. of the 12th Intl. Conf. on Information Processing in Sensor Networks (IPSN)*, 2013.

[22] R. Lim, B. Maag, B. Dissler, J. Beutel, and L. Thiele. A testbed for fine-grained tracing of time sensitive behavior in wireless sensor networks. In *Proc. of the 40th Conf. on Local Computer Networks Workshops (LCN Workshops)*, 2015.

[23] R. Lim, M. Zimmerling, and L. Thiele. Passive, privacy-preserving real-time counting of unmodified smartphones via ZigBee interference. In *Proc. of the 11th Intl. Conf. on Distributed Computing in Sensor Systems (DCOSS)*, 2015.

[24] F. Mokaya, R. Lucas, H. Y. Noh, and P. Zhang. Burnout: A wearable system for unobtrusive skeletal muscle fatigue estimation. In *Proc. of the 15th Intl. Conf. on Information Processing in Sensor Networks (IPSN)*, 2016.

[25] M. Okola and K. Whitehouse. Unit testing for wireless sensor networks. In *Proc. of the on Software Engineering for Sensor Network Applications (SESENA)*, 2010.

[26] C. Pavlopoulou and M. Young. Residual test coverage monitoring. In *Proc. of the 21st Intl. Conf. on Software Engineering (ICSE)*, 1999.

[27] A. Pötsch, A. Berger, G. Möstl, and A. Springer. TWECIS: A testbed for wireless energy constrained industrial sensor actuator networks. In *Proc. of the 19th Intl. Conf. on Emerging Technology and Factory Automation (ETFA)*, 2014.

[28] S. Roman. *Coding and information theory*, volume 134. Springer Science & Business Media, 1992.

[29] P. Sommer and B. Kusy. Minerva: Distributed tracing and debugging in wireless sensor networks. In *Proc. of the 11th ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, 2013.

[30] V. Sundaram, P. Eugster, and X. Zhang. Efficient diagnostic tracing for wireless sensor networks. In *Proc. of the 8th ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, 2010.

[31] M. Tancreti, M. S. Hossain, S. Bagchi, and V. Raghunathan. Aveksha: A hardware-software approach for non-intrusive tracing and profiling of wireless embedded systems. In *Proc. of the 9th ACM Conf. on Embedded Networked Sensor Systems (SenSys)*, 2011.

[32] M. Tancreti, V. Sundaram, S. Bagchi, and P. Eugster. Tardis: Software-only system-level record and replay in wireless sensor networks. In *Proc. of the 14th Intl. Conf. on Information Processing in Sensor Networks (IPSN)*, 2015.

[33] F. Terraneo, A. Leva, and W. Fornaciari. Demo: A high-performance, energy-efficient node for a wide range of WSN applications. In *Proc. of the Intl. Conf. on Embedded Wireless Systems and Networks (EWSN)*, 2016.

[34] Texas Instruments. *MSP430x1xx Family User's Guide*, Feb. 2006. Rev. F.

[35] M. M. Tikir and J. K. Hollingsworth. Efficient instrumentation for code coverage testing. In *Proc. of the 2002 Intl. Symp. on Software Testing and Analysis (ISSTA)*, 2002.

[36] P. Tuset-Peiró, X. Vilajosana, and T. Watteyne. Openmote+: a range-agile multi-radio mote. In *Proc. of the Intl. Conf. on Embedded Wireless Systems and Networks (EWSN)*, 2016.

[37] X. Vilajosana, P. Tuset, T. Watteyne, and K. Pister. Openmote: Open-source prototyping platform for the industrial IoT. In *Proc. of the 7th Intl. Conf. on Ad Hoc Networks (AdHocHets)*, 2015.

[38] L. Wan and Q. Cao. Towards instruction level record and replay of sensor network applications. In *Proc. of the 21th Intl. Symp. on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2013.

[39] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.

[40] M. Woehrle. *Testing of wireless sensor networks*. PhD thesis, ETH Zurich, 2010.