Efficient Diagnostic Tracing for Wireless Sensor Networks

Vinaitheerthan Sundaram
School of Electrical and Computer Engineering
Purdue University
vsundar@purdue.edu

Patrick Eugster
Department of Computer Science
Purdue University
peugster@purdue.edu

Xiangyu Zhang
Department of Computer Science
Purdue University
xyzhang@purdue.edu

Abstract

Wireless sensor networks (WSNs) are hard to program due to unconventional programming models used to satisfy stringent resource constraints. The common event-driven concurrent programming model and lack of kernel protection in these systems introduce the possibility of several subtle faults such as race conditions. These faults are often triggered by unexpected interleavings of events in the real world, and can occur long after their causes. Reproducing a fault from the trace of the past events can play a crucial role in debugging such faults. The same tight constraints that motivate the specific programming model however make tracing challenging. This paper proposes an efficient intra-procedural and inter-procedural control-flow tracing algorithm that generates the traces of all interleaving concurrent events. Our approach enables reproducing faults at a later stage, allowing the programmer to identify them effectively. We argue for the accuracy of our approach through case studies, and illustrate its low overhead through measurements and simulations.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Testing and Debugging—*debugging-aids, tracing*

General Terms

Design, Experimentation, Reliability

Keywords

Embedded Debugging, Tracing, Wireless Sensor Networks, Diagnosis

1 Introduction

Wireless sensor networks (WSNs) require unconventional programming models to satisfy stringent resource constraints. This makes WSNs hard to program. For example, TinyOS/nesC proposes an event-driven concurrent programming model; together with the lack of kernel protection the model makes corresponding WSN applications prone to race conditions. Such defects are often triggered by unexpected interleavings of events in the real world. Therefore, *run-time debugging* tools are required to detect and diagnose these defects in the post-deployment phase.

There have been several debugging solutions proposed for WSNs. (a) Automated debugging tools [19, 13, 11, 15, 6, 8] mostly monitor the network and thus, do not provide much help for programmer errors because causes and symptoms may be far apart. These tools may also impose expensive synchronization to achieve good resilience, or require programmers to express invariants or SQL-like queries to guide run-time monitoring and debugging. (b) "Remote control" approaches [22, 23] try to provide insight into — and control of — remote sensor nodes. These approaches incur a substantial overhead and put much load on the programmer who has to navigate though the program at run-time. Finally, (c) program analysis based tools [18, 9] provide higher-level understanding of programs but fail to quickly pinpoint the exact causes of faults, or exhibit high complexity.

Replay debugging is a powerful run-time debugging technique to diagnose complex faults as it allows reproducing defects by replaying from traces recorded during real execution. It is especially useful for debugging distributed applications because of the inherent non-determinism [4] [14] and has been shown to be effective in WSN macro programming environments [21]. Replaying requires obtaining the trace of the ordered sequence of events or control-flow path taken and the external input values, which may not present insurmountable issues in a wired setting; obtaining it in a WSN can however be prohibitively expensive in terms of bandwidth required to transfer it from a node to the base station. Tight constraints do not only exist on bandwidth, but similarly also on storage resources, which makes it challenging to devise an efficient tracing scheme altogether. The key problem in WSN replay debugging is to decide what information to record while satisfying resource constraints.

For WSN settings, we argue for recording the control-flow information. On the one hand, this information can be effective in fault diagnosis. Despite sophisticated techniques and tools described before, one of the commonly used debugging is LED debugging [20] in which the developer switches LEDs on and off to notify the success or failure of

SenSys'10, November 3–5, 2010, Zurich, Switzerland.

the events of interest. This is similar to `printf` debugging on commodity desktops/laptops. In fact, `printf` capability is one of the most requested features in user mailing lists [20]. The most important information the developer is trying to get by this type of debugging is the control-flow path taken. Knowledge of control-flow exercised when a fault happens can help a great deal in diagnosis for a large number of implementation faults.

Yet, on the other hand, the control-flow information can be captured in a resource-efficient way. One of the important characteristics of WSN applications is the repeated execution of same sequences of actions with occasional occurrences of unusual events. If the control-flow path is properly encoded and recorded, the repetitive sequences of actions can be compressed quite well.

In fact, control-flow traces also capture adequately some of the effects of input values such as sensed values and network messages. For example, if a network message represents a command to the sensor to send current sensed values to the base station, then the control-flow captures the message contents. Likewise, if the sensed value is beyond a threshold, the action taken by the sensor is captured in the control-flow. For other purposes, such as corrupting routing table updates from malicious entities, recording input values could be useful but impractical for WSN settings. Therefore, recording control-flow trace offers a good trade-off that allows partial replay of the execution.

In this paper, we propose a novel efficient tracing technique that encodes and records the control-flow including all the interleaving of concurrent events. To deal with long-running real-life programs, our technique has a low footprint, which is achieved by statically analyzing the program and instrumenting only a few statements inside an event handler. At run-time, the trace is recorded in memory and compressed before being committed to non-volatile external flash memory. When a fault is detected, upon request from a system manager, or at selectively defined points in the program, the trace is sent to the base station for analysis. The programmer can replay the trace in a controlled environment such as a simulator or a debugger to reproduce the fault. In fact, the traces enable reverse debugging of the program, thus allowing the programmer to identify the defect effectively. Our tracing method satisfies the stringent resource constraints of WSNs, by exploiting the specific execution model and making use of compression techniques. While explained and implemented in the context of nesC, our techniques are generalizable.

In summary, the main contributions of our paper are threefold: (1) We present an efficient tracing design that precisely captures the interleaving of concurrent events as well as the control-flow path taken inside the events. (2) As part of the tracing design, we present a novel technique that allows modular computation of inter-procedural control-flow paths. (3) We illustrate the effectiveness of our approach through case studies including a previously unknown bug and demonstrate the efficiency of our solution through performance measurements.

Roadmap. The remainder of this paper is structured as follows. Section 2 presents background on the TinyOS execution model. Section 3 presents an overview of our efficient tracing technique; the details are presented in the following Sections 4, 5 and 6. Section 7 presents the implementation of our tools. Section 8 illustrates the accuracy of our approach through cases studies, and Section 9 dissects its overhead. Section 10 discusses limitations, and Section 11 contrasts our approach with related work. Section 12 concludes with final remarks.

2 Background

The TinyOS event-driven execution model poses unique challenges, but simultaneously allows for efficient solutions compared to the execution model of general purpose program environments allowing arbitrary thread interleavings.

2.1 TinyOS Execution Model

A TinyOS application is often composed of a set of reusable components that are wired through configurations. Components communicate through *interfaces* consisting of *commands* and *events*. A component provides services to others through commands. If a component requests a service, the completion of the request is signaled to the component through an event. Events are also the channels through which the hardware communicates with components.

In TinyOS, there is no explicit thread abstraction because maintaining multiple threads needs precious RAM and thread interleavings easily introduce subtle data race errors. Nonetheless, TinyOS applications need a mechanism for parallel operations to ensure responsiveness and respect real-time constraints. More particularly, low priority operations should give way to critical operations, e.g., interrupts from hardware. In general, there are two sources of concurrency in TinyOS: *tasks* and *event handlers* (also simply called *events* in the following). Tasks and events are normal functions with special annotations. Tasks are a deferred computation mechanism. They run to completion and do not preempt each other. Tasks are posted by components. The post request immediately returns, deferring the computation until the scheduler executes the task later. To ensure low task execution latency, individual tasks must be short. Lengthy operations should be spread across multiple tasks. In contrast, events also run to completion, but may preempt the execution of a task or another event. An event signifies either completion of a lengthy (and thus split) operation or an event from the environment (e.g., message reception, time passing). TinyOS execution is ultimately driven by events representing hardware interrupts.

For example, the operation to get a value from a sensor is often split into multiple phases. First, a command call (down-call) is made to a service component to start the operation, which is carried out by multiple tasks. Later, when the operation is completed, an interrupt occurs, generating a callback (event/up-call) to the consumer component which processes the result in the event handler. In such a process, the tasks of the operation cannot be preempted by any other tasks but can be preempted by events. A key observation is that *the executions of events and tasks are either disjoint or nested.* In contrast, a traditional thread-based concurrency model allows arbitrary thread interleaving.

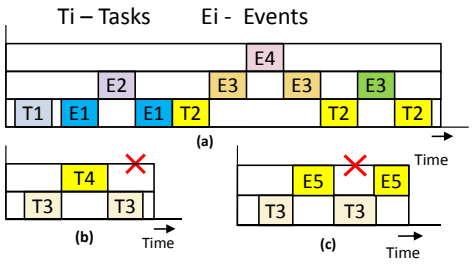


Figure 1. TinyOS Model. Boxes depict execution of tasks or events. The y-axis represents the preemption level.

Figure 1 shows a snapshot of how tasks and events execute in the TinyOS concurrency model. Boxes with the same color in Figure 1 represent the execution of a task/event and their labels denote the name of the task/event. Observe that a task or an event may be preempted, giving rise to multiple boxes with the same color. The level of nesting involved in preemption is shown vertically for clarity. Figure 1(a) presents a legal execution. Task *T1* is executed without preemption. *E1* occurs sometime after *T1* and is preempted by event *E2*. Once event *E2* finishes, event *E1* resumes and runs to completion. Task *T2* and events *E3* and *E4* represent a more complex case in which multiple event preemptions occur. When task *T2* is running, event *E3* occurs and preempts task *T2*. Event *E4* occurs and preempts event *E3* and runs to completion, upon which *E3* resumes and runs to completion too. Task *T2* resumes and gets preempted by another invocation of event *E3*, which may correspond to another instance of the same hardware interrupt is received by the handler *E3*. Once *E3* completes, *T2* again resumes and runs to completion without further preemption. Figures 1(b) and (c) represent impossible sequences of executions. In Figure 1(b), a task cannot preempt another one. In Figure 1(c), the executions of a task and an event cannot interleave as illustrated because the preempting execution has to complete before the preempted execution gains control.

Finally, the executions of tasks and events are not simply the instantiations of the functions with the task/event annotations as they can invoke other functions. Invoked functions may even be recursive. These cases need to be handled.

3 Efficient Tracing Design Overview

We propose an efficient tracing technique that is suitable to resource-constrained event-driven systems such as TinyOS applications. Our key idea is to represent the interprocedural control-flow of tasks and events with few bytes and record them as they are executed. Since TinyOS applications execute the same tasks and events repeatedly, our representation enables significant compression of the traces.

We present our technique in several steps starting with recording at the level of tasks and events and then, go down gradually to reveal the full design. In Section 4, we describe our method to record interleaving of tasks and events by exploiting the concurrency model outlined earlier. In Section 5, we first motivate the need to record control-flow paths. Then, we describe our method to record the intra-procedural paths within a task or event, assuming there are no function calls in tasks or events. We then relax that assumption and describe a novel modular technique to compute interprocedural paths which reduces the trace size and CPU cycles used. We also compare it to the state-of-the-art techniques. In Section 6, we describe our generic compression algorithm.

$$Trace \rightarrow EUnit* \qquad EUnit \rightarrow Id\ Event*\ \textbf{\textit{end}} \mid \varepsilon$$
$$Id \rightarrow \textbf{\textit{Tid}} \mid \textbf{\textit{Eid}} \qquad Event \rightarrow \textbf{\textit{Eid}}\ Event*\ \textbf{\textit{end}} \mid \varepsilon$$

Figure 2. Task/event level grammar. *Tid* and *Eid* are identifiers for tasks and events, respectively. *EUnit* represents an execution unit, i.e. an instance of a task or event. *end* is a special symbol denoting the end of a task or an event.

4 Task- and Event-Level Tracing

We are first interested in the tasks and events that are executed and their interleavings but not the detailed execution paths inside those. Complex failures of TinyOS applications are often due to unexpected interaction of different interrupt handlers. For example, the receiver buffer being simultaneously modified by an application and the network layer is a common but difficult to diagnose fault in WSN programs [8]. To diagnose such complex failures, information on interleaving is essential.

4.1 Challenges

The event-based concurrency and deferred execution of tasks present unique challenges in recording interleavings as interrupts can occur at any time in a program and the handling of interrupts can create tasks. Furthermore, sensors feature very limited computation resources, mandating a cost-effective design.

A naïve design is to record the identifiers of each executed code block. For example, the execution in Figure 1(a) can be represented by the following trace:

T1 E1 E2 E1 T2 E3 E4 E3 T2 E3 T2

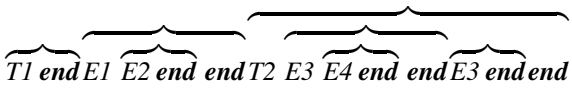
This has severe limitations. First, the nesting structure is not properly captured. One can not tell that the first two *E3*'s belong to the same instance and the third *E3* is a standalone one, whereas the three *T2* blocks belong to the same instance. Second, the trace is redundant as encodings can be inferred. For instance, the second *T2* is unnecessary as when *E3* completes, the execution preempted by *E3* is supposed to resume so that *T2* must be the continuation of *E3*.

4.2 Approach

The nesting structure of TinyOS application execution can be exploited such that traces can be described by a context-free grammar. The grammar is presented in Figure 2.

The intuition of the grammar is as follows. A trace is composed as a sequence of execution units (*EUnit*), which could be a task or an event. An *EUnit* is delimited by its identifier *Id* and a universal ***end*** symbol. The fact that a task and an event can be preempted by events is represented by the *Event* Kleene-closure in the rule of *EUnit*.

The execution in Figure 1 can be depicted by the following string of the grammar. Overbraces represent the nestings.



The encoding not only captures the nesting, which allows understanding of interleavings, but also needs fewer identifiers (reduced from 11 to 7) than the naïve approach described earlier. Although a number of ***end*** symbols are required, the symbol is universal so that much fewer bits are needed to encode compared to task/event identifiers.

Context-free grammars can be parsed by push-down automata. Thus, the trace can be received and then parsed by the base station that has more computational resources, using a stack. The nesting structure of the trace is naturally constructed by the parse tree. The redundant identifiers that are removed from the naïve trace, e.g., the second and third *T2* blocks, can be reproduced during parsing by inspecting the stack's state. The parsing algorithm is omitted for brevity.

5 Tracing Inside Tasks and Events

Task- and event-level interleavings are insufficient for debugging in most cases. In this section, we motivate the need to record instruction-level control-flow information. We also present a design that can cost-effectively achieve this goal. We first discuss how to record the control-flow path in the simple case in which tasks and events do not have function calls. Then, we allow function calls and present a novel modular technique to compute inter-procedural paths which reduces the trace size and CPU cycles used compared to the state-of-the-art techniques.

5.1 Challenges

The task- and event-level traces generated in Section 4 capture high-level interleavings. However, they are insufficient because they do not have fine-grained control-flow information within the tasks/events and also, do not contain the exact preemption points. Different preemption points often lead to different program state and hence different control-flow if the interleaved executions access some shared variables. As reported in [8], a large portion of TinyOS application faults fall into this category of incorrect shared access. In Section 8.2.1, we discuss one such fault.

Preemptions are directly supported by hardware and thus, transparent to TinyOS. By contrast, in regular systems with a different concurrency model, the OS is aware of context switches as the switches are performed by utilizing OS services. One possible solution is to instrument all interrupt handlers to read the return address off the stack and record it into the trace. This approach is platform-specific as ATMEL's ATMEGA128 and TI's MSP430 store return addresses at different stack offsets. Another possible solution is to instrument TinyOS applications so that the program counter of each executed instruction can be recorded. Hence, a preemption can be identified from the trace as an unexpected program counter alternation. Clearly, such a solution is very expensive as a non-trivial instrumentation has to be inserted for each instruction.

5.2 Approach

We propose a technique to trace inside tasks and events and record the control paths of their executions. In our design, the preemption points can be more precisely located. More importantly, by knowing the exact sequence of executed instructions, the effects of taking these preemptions are recorded, which is often sufficient for debugging. For example, in the case where different interleaving induce different conditional branches being taken, the control-flow trace precisely captures the effect of the interleaving by retaining which branch was taken. The challenge of our design thus lies in efficiently representing control-flow paths.

One may think that it is better to record input values such as messages received or sensed values. In other words, replay could be achieved by using the recorded input values and the high-level interleaving trace. However, in such a design, neither preemption points nor their effects are recorded such that the execution cannot be easily and faithfully reproduced. Furthermore, compared to our efficient control-flow path tracing design, precisely recording inputs could be much more expensive as some inputs such as network messages are long and are not as highly compressible as control-flow paths.

5.3 Intra-procedural Control-flow Tracing

For the sake of presentation, we first explain our design without considering function calls. In other words, we assume that execution units do not have function calls. The basic idea is to encode intra-procedural control-flow paths. More specifically, we represent a control-flow path with an integer from 0 to $n-1$ with n being the total number of intra-procedural paths of a given procedure [1].

5.3.1 Background

In their seminal paper, Ball and Larus [1] proposed an efficient algorithm (referred to as the BL algorithm) to compute the optimally encoded intra-procedural control-flow path identifier taken during execution. The BL algorithm features translating acyclic path encoding to instrumentations on control-flow edges. At run-time, a sequence of these instrumentations are executed following a control-flow path, resulting in the path identifier being computed. All these instrumentations involve only simple additions. The idea can be illustrated by the example in Figure 3. The code is shown on the left and the control-flow graph is shown on the right. The instrumentations are marked on control-flow edges. Before the first statement, `lcount` is initialized to 0. If the false branch is taken at line 1, `lcount` is incremented by 2. If the false branch is taken at line 5, `lcount` is incremented by 1. As shown on left bottom, executions taking different paths lead to different values in `lcount`. In other words, `lcount` encodes the path.

The basic algorithm is presented in Algorithm 1. The algorithm first annotates each node with the number of paths that lead from that node to the `Exit` of the procedure. In an acyclic graph, the annotation of a node can be computed by summing the annotations of its children. For instance in Figure 3, node 1 has the annotation of `4`, which is the sum of the annotations of nodes 2 and 4, meaning there are 4 paths leading from 1 to `Exit`. The instrumentation on an edge $n \rightarrow m$

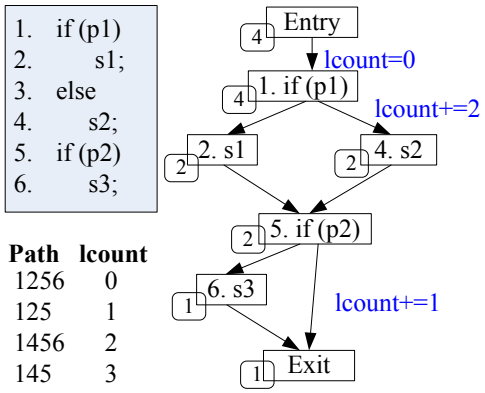


Figure 3. Example for Ball-Larus path encoding. Instrumentations are marked on control-flow edges. Node annotations (rounded boxes at the corners) represent the number of paths leading to `Exit`.

Algorithm 1 The BL Algorithm.

annotate each node with the number of paths rooted at the node
for each edge $n \rightarrow m$ **do**
  $s \leftarrow 0$
  for each edge $n \rightarrow t$ and t precedes m in n's edge list **do**
    $s \leftarrow s$+t's annotation
  end for
  instrument $n \rightarrow m$ with "`lcount+=` s"
end for
instrument the exit node with "`trace.print(lcount)`"

is computed as increasing the path id counter `lcount` by the sum s of the annotations of all the children of n that precede m. Intuitively, it means the encoding space from `lcount` to `lcount`+s-1 is allocated for the paths following the edge from n to some child before m. Note that there are s such paths. The paths following edge $n \rightarrow m$ use the encoding space that is greater than `lcount`+s-1. For example, the instrumentation on $1 \rightarrow 4$ is "`lcount+=2`" as the annotation of 1's preceding child 2 is 2, meaning the range `lcount` $\in$ [0,2) is allocated to the paths following the edge $1 \rightarrow 2$ and `lcount` $\in$ [2,4) is allocated to the paths following $1 \rightarrow 4$. Finally, the algorithm instruments the end of a function by emitting the path id to the trace.

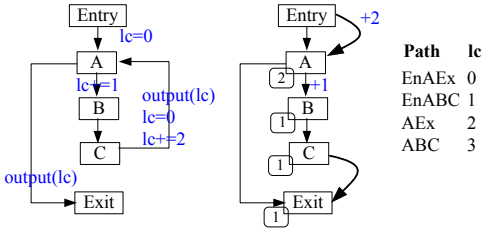


Figure 4. Example for loop path encoding. Variable `lc` holds the path identifier, which is emitted at back-edge and procedure exit. The subgraph on the left shows the program and the instrumentation, which is computed from the transformed graph on the right. In the transformed graph, the back-edge is replaced by two dummy edges as highlighted.

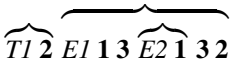
Loop paths are handled by dividing them into acyclic paths that end at a back-edge. More particularly, the cyclic control-flow graph is transformed into an acyclic one by removing back-edges and introducing dummy edges from `Entry` to the loop head and the last node in the loop body to `Exit`. Figure 4 shows an example. The original graph is shown on the left and the transformed (acyclic) graph is shown on the right. Note that the transformed graph is only used for computing path id increments, the program itself is not transformed. The path id increments are computed in the same way as before. The key idea is that the increments on the dummy edges are translated to instrumentations on the back-edge in the original program as shown on the left. In the example, the back-edge instrumentation first emits the current path id to the trace as it encounters the end of a loop path. It then resets the counter. The increment by 2 after that is due to the increment on the dummy edge from `Entry` to `A`. One can observe that these instrumentations generate the encodings of acyclic paths on the right. In other words, the trace of "`Entry A B C A B C A Exit`" is encoded as "`1 3 2`". The detailed algorithm can be found in [1]. Note that the existence of loop paths leads to the control-flow of an execution unit being represented by a sequence of ids instead of just one id.

5.3.2 *Bit Tracing Comparison*

It is possible to trace the control-flow by recording one bit for every branch. However, this has several disadvantages. As mentioned in [1], such a tracing is not optimal in terms of encoding space (a counter-example can be easily constructed) and requires instrumentation on every branch edge. In addition, bit-tracing requires the trace from the beginning of the execution if we need to replay the trace and does not handle concurrency unless special symbols are recorded.

5.3.3 *Intra-procedural Control-flow Trace Grammar*

Now, we extend our high-level execution trace grammar to include the control-flow information using the BL encoding. We call this *intra-procedural* control-flow tracing. In this trace, the control-flow path traversed is recorded in between the execution unit id and the corresponding ***end*** symbol. Note that the execution units that have loops would record a sequence of ids representing the paths during each iteration. For example, let task *T1* and event *E2* have no loops and *E1* be the program in Figure 4 and have a loop. In the following intra-procedural control-flow trace recorded, we show the paths traversed in the task *T1* and events *E1* and *E2*. The ***end*** symbol is omitted from the trace for clarity. We note that the event *E1* has executed three loop iterations and the interrupt happened after the second iteration. Observe that the trace not only records the detailed paths taken, which may be the consequence of the interleavings, but also narrows down the places where the preemption occurs, which must be inside the third iteration.



We observe that the intra-procedural control-flow trace still retains the nested structure in TinyOS applications and can be described by a context- free grammar. The grammar

$$
\begin{array}{rcl}
Trace & \rightarrow & EUnit* \\
EUnit & \rightarrow & Id\ (Event|Path)*\ Path\ \textbf{\textit{end}} \mid \varepsilon \\
Id & \rightarrow & \textbf{\textit{Tid}} \mid \textbf{\textit{Eid}} \\
Path & \rightarrow & \textbf{\textit{Pid}} \\
Event & \rightarrow & \textbf{\textit{Eid}}\ (Event|Path)*\ Path\ \textbf{\textit{end}} \mid \varepsilon
\end{array}
$$

Figure 5. Intra-procedural control-flow trace grammar. *Pid* is identifier for paths

$$
\begin{array}{rcl}
Trace & \rightarrow & EUnit* \\
EUnit & \rightarrow & Id\ (Func|Event|Path)*\ Path\ \textbf{\textit{end}} \mid \varepsilon \\
Id & \rightarrow & \textbf{\textit{Tid}} \mid \textbf{\textit{Eid}} \\
Path & \rightarrow & \textbf{\textit{Pid}} \\
Func & \rightarrow & \textbf{\textit{Fid}}\ (Func|Event|Path)*\ Path\ \textbf{\textit{end}} \mid \varepsilon \\
Event & \rightarrow & \textbf{\textit{Eid}}\ (Event|Path)*\ Path\ \textbf{\textit{end}} \mid \varepsilon
\end{array}
$$

Figure 6. Inter-procedural control-flow trace grammar. *Fid* is identifier for functions. A new nonterminal *Func* is introduced to represent function calls.

is presented in Figure 5. The intuition of the extension to our earlier grammar is as follows. An *EUnit*, delimited by its *Id* and an universal ***end*** symbol, must include a control-flow path just before the end and can include any number of events that preempt the *EUint* or any number of loop paths. Since events can be preempted and can have loops, *Event* has the same RHS. Observe that a similar predictive top-down parser can be constructed for the grammar as earlier.

5.4 Inter-procedural Control-Flow Tracing

The primary execution units (tasks/events) may call functions and the control-flow path of an execution unit should include the control-flow path taken inside the called functions. Note that tasks and events can be viewed as functions that are not invoked by a caller. In order to make the distinction though, we refer to them as *execution units* and all others as *function calls*.

5.4.1 Challenges

Since function calls exhibit the same nesting property as execution units, i.e., a called function's execution is completely nested within the caller's execution, a straightforward solution consists in treating function calls the same way as execution units. More particularly, as shown by the context-free grammar in Figure 6, a function call, denoted by *Func*, can reside in the *EUnit*, *Event*, or *Func* itself. The last case describes a function calling another function.

For example, consider the following string of the grammar in Figure 6. In this trace, we see that the event *E1* calls function *F* which in turn calls function *G*. Event *E2* preempts *F*'s execution. The nested structure of execution units and function calls are correctly captured in the trace.

The problem with the simple approach is that TinyOS applications often have a large number of small functions. According to the grammar in Figure 6, a function id, a special ***end*** symbol and at least, one path id are needed for each function invocation even though the invocation execution may be very short. One possible solution is to inline small functions. However, this will substantially increase the code size, which is especially not affordable in resource-constrained WSNs.

5.4.2 Approach

Our approach focuses on representing the inter-procedural control-flow path of the entire execution unit with a single id instead of a sequence of ids for individual function calls. This approach can reduce the number of bytes recorded in the trace significantly as only one id is needed to encode the exact path being traversed through multiple function calls.

The challenge lies in that the different invocation points of a function demand different versions of instrumentation for the function in order to produce the correct inter-procedural path encoding, which depends on the invocation points. For instance, in Figure 7, there are two invocation sites of function *Foo*. They are *B* and *F*. In order to produce correct inter-procedural encoding regarding call site *F*, the path id counter should be increased by 1 along the edge $J \rightarrow L$ because there is only one path leading from *L*'s left sibling *K* to the end of the inter-procedural path, which is *G*. However, when the call site *B* is considered, since in the context node *K* has 4 paths (partially induced by the predicate inside *Foo* called at *F*), the instrumentation should increase the path id by 4 along the edge $J \rightarrow L$. Such context-sensitive instrumentation is hard to achieve.

Our solution is presented as follows. Let n and p denote the total number of paths inside the callee function and the exact path taken inside the callee function at run-time, respectively. Let x refer to the number of paths to exit from the call site's successor in the caller function. Note that the value of n and x can be obtained statically and the value p is returned by the function at run-time. *The key idea of our technique is to have only one version of instrumentation for a function. The context-sensitivity is handled by adjusting the caller's run-time path id by leveraging the values of n, p, and x.* More particularly, we annotate the call-site node with the product of n and x ($n \times x$) because there are $n \times x$ (inter-procedural) paths leading from the call-site to the end as every path inside the called function can be followed by any one of the x paths after the function call.

The edge between the call-site node and the successor is annotated with the product of p and x ($p \times x$). Intuitively, the multiplication amplifies the encoding space from the callee by a factor of x to allow encoding the x paths following the call site. More particularly, if the path inside the callee is p, then the interval $[p \times x,\ (p+1) \times x)$ is used to encode the inter-procedural paths led by the callee path p and trailed by one of the x paths in the caller.

The inter-procedural instrumentation algorithm is presented in Algorithm 2. The algorithm instruments a given function F, taking care of the encoding of the functions called by F.

We assume that the functions called inside F, referred as G in Algorithm 2, do not have loops or recursions. Otherwise, the simple individual function encoding will be employed. Note that F itself can have loops. We explain the

benefits of using individual function encoding when there are loops and recursion in the called functions in Section 5.4.4.

Algorithm 2 Instrumenting a function F without loops or recursion. Assume all function invocations have been made independent statements instead of sub-expressions as in a low-level intermediate representation, i.e., each call site has only one successor. Function `retrievePath` retrieves the path id of the callee at run-time.

```
for each node s in F's CFG in the reverse topological order do
  x ← the sum of the annotations of s's successors
  if s is a function invocation then
    G ← the function invoked at s
    if G does not have loops or recursion then
      n ← the number of static paths of G
      s.annotation ← s × x
      instrument edge s → s.succ[0] with "lc+=x×retrievePath(G)"
    else
      s.annotation ← x
    end if
  else
    s.annotation ← x
    for i = 1 to the number of s's successors do
      sum ← the sum of s.succ[0−(i−1)].annotation.
      instrument s → s.succ[i] with "lc+=sum"
    end for
  end if
end for
if F is a task or event then
  instrument the entry node with "trace.print(id)"
  instrument the exit node with "trace.print(lcount)"
end if
```

Similar to Algorithm 1, the annotation of a node represents the number of (inter-procedural) paths from the node to the end of the function. The algorithm traverses each node in the CFG in a reverse topological order. If the node represents a function invocation and the function being called, denoted as G, does not have loops or recursion, the paths inside G will be encoded as part of the path id of F in a fashion as explained earlier. The call to `retrievePath` in the instrumentation is provided as part of the tracing library to get the aforementioned p, the run-time path inside G. More particularly, the call retrieves the variable `lc` of G, which is a local variable on G's stack frame, encoding the path just taken inside G. Note that the value is not destroyed as the execution just returns from G. At the end, the path id is emitted to the trace if and only if F is a task/event, i.e., a top level function invocation. Otherwise, the path id is expected to be retrieved by the caller of F.

5.4.3 Illustration

We use the example in Figure 7 to illustrate the idea. Each node is annotated with the number of inter-procedural paths from that node to the exit. The annotations are shown as rounded boxes at the corners. The edges are annotated with path id increments.

The function *Foo* has two paths namely α and β and the value of n is 2. The number of paths to exit from F's successor is 1 ($x = 1$) and therefore, the number of paths from F to exit is 2 ($n \times x = 1 \times 2$), which is shown as F's annotation. Now this value is propagated upwards in the control-flow graph of *Main*. The number of paths to exit from B's successor, C is 4 ($x = 4$) which includes the two possible

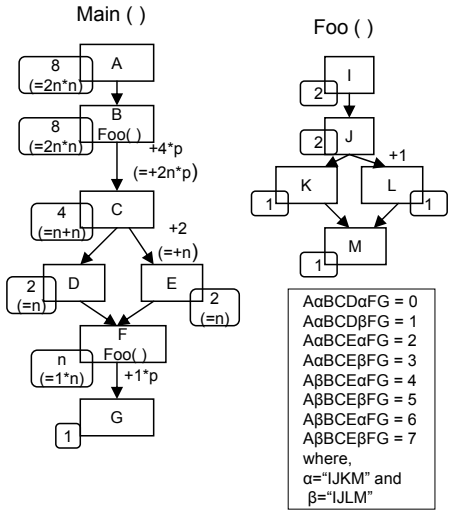


Figure 7. Example for the modular inter-procedural paths computation algorithm.

paths inside function *Foo* called at F. The annotation on call site node B is therefore 8 (= 4×2). The edge increments for edges $B \rightarrow C$ and $F \rightarrow G$ are annotated with $4 \times p$ and $1 \times p$, where p can be 0 or 1 depending on the path inside *Foo* taken at run-time. The set of inter-procedural paths and their encodings are shown in the figure.

5.4.4 Loops and Recursion

Let the loop path refer to the control-flow path that ends at loop exit. We observe that the loop path can be inter-procedural if the loop is in a called function. It is important to use fewer bits to record the loop path, i.e., each iteration of a loop, to reduce the trace size. Our approach to reduce the number of bits to record loop paths is to capture the context of the loop once and create a local namespace. The context is captured by recording the start and end of the called function that has loops. The idea of using a local namespaces for logging has shown to be advantageous in the context of WSN settings [20].

If we do not capture the context, the global namespace has to be used for encoding all the loop paths in the program. The advantage of this approach is that every path including the loop path is globally unique. However, since the inter-procedural paths include the transitive closure of the called functions, the global namespace may become very large. Larger namespaces mean using more bits per loop iteration recorded.

Melski and Reps [16] extended the BL algorithm, referred to as the MR algorithm, to build a super graph, i.e., an inter-procedural control-flow graph, and compute all edge increments in that super graph. However, unlike the BL algorithm, *the MR algorithm's edge increments are linear functions that involve multiplications because the edge increments inside a function depend on the call sites of the function.* In order to handle such cases, the MR algorithm uses function parameters to pass context-sensitive coefficients of the linear functions. In other words, different coefficients are

used for different contexts.

In comparison, we have only one version of increments for the function being called and only one multiplication is needed to adjust the path id of the caller at the call site. Having fewer expensive instructions is particularly important for sensor programs. Our algorithm is much simpler to implement and also takes advantage of local identifiers for repeating loop paths, thus saving on bits required to encode the paths and less CPU to calculate the identifiers. We show in Section 9.4, that the trace size of the MR algorithm is 14% to 35% more than that of our algorithm when the identifier uses one more byte than ours.

A disadvantage of maintaining inter-procedural path in the context of tracing or debugging is that if we do not record the paths often to persistent storage we may lose the entire execution if the node gets restarted.

In WSN applications developed using TinyOS, we found that the number of iterations in most loops is small and the control-flow paths inside the loops are trivial (like packet copy, doing something for each neighbor or waiting for a device). This indicates that we can apply optimizations such as loop unrolling or compressing the loop entries if there is only one path in the loop.

6 Generic Compression

In TinyOS application executions, a path is often exercised many times, giving rise to a large number of repeated subsequences in the trace. Such repetition is hardly exploited by the encoding scheme we have presented so far, which relies on program structure instead of run-time patterns. For instance, our acyclic path encoding entails that a loop path has to be divided into paths for individual iterations and then recorded as a sequence of path ids, even though these ids may be the same. For normal scenarios where the resources are not so constrained, the repetition can be easily captured by using `zlib`, which implements the LZW compression algorithm, or `Sequitur` [12]. However, they are way too expensive for WSNs.

Fortunately, the repetition patterns are simpler in our case because of the repeating sequences of events as well as the small numbers of unique acyclic paths inside loops. Therefore, we use a simple table lookup algorithm to further reduce the trace size. In particular, we trace the program offline and identify the most frequent subsequence in that trace. In our benchmarks, the size of the most frequent subsequence varied from 8 to 26 entries. We replace the occurrences of this subsequence with a special symbol in the trace and identify the second-most frequent subsequence in the compressed trace. This second-most frequent subsequence might contain the special symbol of the most frequent subsequence.

We opted for this approach as opposed to mining patterns on the go to reduce the run-time overhead and related memory overhead. As shown in Section 9.3, this compression is quite effective and incurs much less overhead compared to LZW or other similar compression techniques that mine patterns on the go.

During run-time, we use two trace buffers. When one trace buffer is full, the trace generated is recorded in another buffer and a `CompressAndStoreTrace` task is posted on the full buffer. This task replaces all the occurrences of the most frequent subsequence with a special symbol. Then, it does the same for the second-most frequent subsequence. After replacing with the symbols, it does run-length encoding of the repeating loop paths in the trace. Once the compression is done, the buffer is written to flash.

```
1  <OscilloscopeM__Timer__fired> start
2  <OscilloscopeM__Timer__fired> end 0
3  <OscilloscopeM__ADC__dataReady> start
4  <OscilloscopeM__ADC__dataReady> end 3
5     ... [the 4 lines above repeats 9 times]
6  <OscilloscopeM__dataTask> start
7    <LedsC__Leds__yellowToggle> start
8    <LedsC__Leds__yellowToggle> end 1
9  <OscilloscopeM__dataTask> end 0
10 <OscilloscopeM__DataMsg__sendDone> start
11 <OscilloscopeM__DataMsg__sendDone> end 0
```

Figure 8. Partial listing of pretty-printed trace

7 Implementation

We implemented our algorithms in nesC compiler version 1.3 for mica2 motes running TinyOS1.x. We used the CIL source-to-source transformation tool for C to instrument the C code generated by the nesC compiler. The instrumented code is then compiled with `avr-gcc` to create the executable that can run on mica family of motes as well as on simulators Atemu and Avrora. The trace is recorded in the flash at run-time and can be retrieved for later use. We also developed a trace parser using Python that can uncompress and pretty-print the trace. An example of a pretty-printed trace is shown in Figure 8.

Our implementation has two core components: a compile-time CIL module that does inter-procedural analysis and automatically instruments the code, and a run-time TinyOS component, `CFTracerM`, that records the trace.

`CFTracerM` is implemented as a nesC component which has two trace buffers, each of length 192 bytes and 16 flash pages, each of length 16 bytes. `CFTracerM` has a task called `CompressAndStoreTrace` that runs in the background, compresses the buffer, and stores the compressed buffer into flash pages, which are drained by the `EEPROM` component. It is important to implement this task as a state machine that does some useful work and posts itself repeatedly, as we noticed long-running tasks can starve other executions including trace storage. Since the trace is constantly generated, `CFTracerM` has to carefully coordinate the buffer usage with locks. The buffer sizes are configurable and depend on the application being instrumented. For most programs, a trace buffer of size 96 bytes and 8 flash pages is enough to capture all the trace generated by the application.

8 Case Studies of Common Faults

In this section, we identify four common faults that occur in TinyOS/NesC type of systems, namely, initialization faults, split-phase faults, state machine faults, and task queue overruns. These faults have been reported several times in the literature [23, 7] as well as in mailing lists. They also played a crucial role in the design of TinyOS2.x. We discuss how our tracing scheme can aid in diagnosing these common faults and also share our experience in using our tool to

debug two of these faults which occurred in our implementation. We uncover one previously unknown bug in the widely used flash/EEPROM component in TinyOS 1.x.

Wherever applies, spurious code statements contributing to the defect are followed by "`[delete]`"; missing statements to be inserted to repair the defect are headed by "`[insert]`'.

8.1 Initialization Faults

Forgetting to initialize the components is one of the top faults in TinyOS 1.x and therefore, in TinyOS 2.x, the bootloader calls the initialization function on all components by default. The manifestation of this fault can be subtle like we discuss below through one such manifestation in TOSSIM.

8.1.1 TOSSIM Core Dump

This fault was experienced by us when we were developing and evaluating our tool. The bug was due to incorrect usage of our tool. We forgot to wire the `StdControl` interface provided by the EEPROM (flash) to the main component and hence, EEPROM was not correctly initialized. The flash component is referred to as EEPROM in the TinyOS 1.x code and documentation. The manifestation of this fault was subtle and interestingly, causing the TOSSIM simulator to crash instead of inducing a flash failure.

8.1.1.1 Fault Description

TOSSIM dumped core after running one of the benchmark program for a while and exited. Since TOSSIM is a discrete event simulator, `gdb` stack trace shows only the functions in the event loop and does not point to the function that inserted the null simulator event. The debug output showed that the program just finished executing the trace compression task and we could not find any problem with that task.

8.1.1.2 Tracing

We turned on tracing on all components used, including radio and flash, and our tracing tool printed the trace to a file in TOSSIM. The trace indicated that a write to EEPROM was attempted just before the compression task execution but was not completed. Interestingly, though, the EEPROM `startWrite` and `write` functions took the successful control-flow path but the `writeDone` was never executed. This raised the suspicion on EEPROM `write` and we noticed that `write` inserts a simulator event, which has a null handler. The handler was assigned when the EEPROM was initialized. Since we forgot the wiring of initialization of EEPROM, the handler had the default null value in it.

Since the event inserted by the EEPROM executed in the future, and in the meanwhile other events and the compression task executed, the debug/`printf` output did not point to the error.

The diagnosis would have been simple had the EEPROM failed at the `startWrite`, as the debug output would have pointed to the EEPROM component directly. The diagnosis became tricky because the EEPROM has a fault that has not been uncovered as far as we know. The state machine used in the EEPROM starts in the idle state instead of an uninitialized state as shown in Figure 9. The programmer has checked for idle state before starting a write, however, did not anticipate that the system could be in idle state bypassing the regular initialization scheme. The problem is due to the **`enum`** used to maintain state which by default assigned a value of zero. To fix the bug, an additional state *uninitialized* has to be created as shown in the Figure 9, which prevents any uninitialized access to the component.

```
1  enum { // states
2     S_IDLE = 0,   // <-- [delete]
3          S_UNINIT = 0,  // <-- [insert]
4          S_IDLE = 5,   // <-- [insert]
5     S_READ=1, S_WIDLE=2, S_WRITE=3,S_ENDWRITE = 4
6   };
7   command result_t StdControl.init() {
8     state = S_IDLE;
9     return call PageControl.init();
10  // [creates TOSSIM event]
11  }  ...
12 command result_t EEPROMWrite.startWrite[uint8_t id]() {
13    if (state != S_IDLE) // [fault is executed]
14      return FAIL;
15    state = S_WIDLE;
16    ...
17    return SUCCESS;
18  }
19  command result_t EEPROMWrite.write[uint8_t id](uint16_t
20    line, uint8_t *buffer) {
21    if (state != S_WIDLE || id != currentWriter)
22      return FAIL;
23    if (call PageEEPROM.write(line ... )== FAIL)
24      return FAIL; ...
25  }
```

Figure 9. Partial listing of Flash/EEPROM component in `tos/platform/mica/eepromM.nc`

Most of the state-of-the-art debugging techniques are not applicable to this bug except for function tracing techniques such as NodeMD [11] and LIS [20], which could have helped with much programmer effort.

8.2 Split-Phase Faults

Split-phase operations are resource-efficient in stack usage but using such operations is tricky as the programmer has to manage state across the start and end of the operation manually. This can lead to implementation faults such as high-level data races despite the use of nesC atomic operations.

8.2.1 High-Level Data race

We describe a subtle fault due to a high-level data race which occurred in an implementation of the LEACH protocol [5].

8.2.1.1 LEACH

LEACH is a TDMA-based dynamic clustering protocol. The protocol runs in rounds. A round consists of a set of TDMA slots. At the beginning of each round nodes arrange themselves in clusters and one node in the cluster acts as a cluster head for that round. For the rest of the round, the nodes communicate with the base station through their cluster head. The cluster formation protocol works as follows. At the beginning of the round, each nodes elects itself as a cluster head with some probability. If a node is a cluster head, it sends an advertisement message out in the next slot. The nodes that are not cluster heads on receiving the advertisement messages from multiple nodes, choose the node closest to them based on the received signal strength as their cluster head and send a join message to that chosen node in the next slot. The cluster head, on receiving the join message, sends a TDMA schedule message which contains slot

allocation information for the rest of the round, to the nodes within its cluster. The cluster formation is complete and the nodes use their TDMA slots to send messages to the base station via the cluster head. After sending the TDMA schedule message, each cluster head sends a debug message to the base-station informing the nodes in its cluster.

8.2.1.2 Fault Description

The implementation of LEACH was using a finite state machine, in which the state represents the stage the protocol was executing in, e.g., `SEND_TDMA_SCHEDULE`. The implementation on TOSSIM was run using CC1000 bit-level radio. When the number of nodes is small (less than 20), the implementation worked well. However, once the number of nodes exceeded 50, the amount of data received in the base station started to go down significantly. When looking at the debug logs printed, it became apparent that many nodes did join the cluster but still did not receive a TDMA schedule message from their cluster head and therefore, did not participate in the rest of the round. However, the debug logs at the cluster heads showed successful sending of TDMA scheduling messages.

Since MAC layer acknowledgments were enabled, successful sends indicated successful receipts. It was not clear at that point why the network layer dropped the packet.

8.2.1.3 Tracing

When tracing was enabled on both the sender and the receiver components, the sender turned out to be the culprit. When the load was high, *the trace indicated that there was a timer event fired between the TDMA schedule message send and the corresponding* `sendDone` *event in the cluster head*. The timer corresponds to the state transition timer which fires at the beginning of each slot. The cluster head moved into the next state and tried to send a debug message but did not succeed as the radio send flag was busy indicating another send being in progress. The trace discloses that before checking the radio sending flag, it modified the message type. Since the message buffer is shared, the TDMA schedule message type was modified into a debug message type.

The nodes in the cluster dropped this message after seeing the type, which is intended only for the base station. This error manifested only when the number of nodes was increased because the increase in load caused the TDMA schedule message to be retried several times and the original timeslot was not enough for the message transmission. This error would have been challenging to localize given that it occurs only at high load with the particular interleaving. It is not clear how the state-of-the-art debugging techniques could have helped without capturing both nesC's interleaving of tasks and events as well as the control-flow inside events.

This is a high-level data race and nesC cannot flag it. If the application modifies the message through a pointer while the message is being sent by the network layer, nesC cannot detect the race because nesC's static analysis does not find racing variables that are accessed through pointers [3]. Thus, tracing the control-flow can be very helpful in tracking high-level data race conditions such as the one explained above.

8.3 Overrunning Task Queue

Task queue overruns represent a class of notorious defects in TinyOS which has caused serious problems including continuous soft resets [7] and deadlocking of the radio stack [23]. Task queue overflow can happen if tasks are starved from execution due to a long-running task [7] or high rate of interrupts [23]. In the interest of space, we just describe how our technique could have helped diagnose the problems.

8.3.1 PermaSense

The cause of the bug was a lookup task whose running time increases with the data stored in the external flash memory. After several months of deployment, that task execution time became large enough that starved other tasks from executing, resulting in a task queue overrun and causing a soft reset, which reinitializes memory and restarts the application. Since the task execution time depended on the external flash, the fault survived a soft reset and continued to cause more resets. It took several months before the fault was diagnosed. If the control-flow tracing were turned on after the first soft reset and the trace were collected until the next soft reset, the control-flow trace would contain the look up task and the repeated interaction with the external flash device. This would have hinted the problem.

The state-of-the-art of debugging techniques such as Clairvoyant [23] could have helped diagnosing this fault but it is not clear how much messaging overhead would have been involved.

8.3.2 CC1000 Radio Deadlock

The CC1000 Radio Stack had a subtle fault that led to deadlocking. This fault has been discussed in the mailing lists, and Yang et al. [23] showed how Clairvoyant could have helped diagnose this tricky fault. Essentially, the fault is due to a small delay before posting tasks in the SPI interrupt handler. The SPI interrupt handler rate was so high that the interrupt handler starved the task queue. If we applied our control-flow tracing technique to the CC1000 radio component execution, the trace would contain the repeated invocations to the SPI interrupt handler with the path traversed denoting the task queue overrun.

8.4 Faulty State Machines Implementations

Event-driven programming relies heavily on finite state machines to program sequences of actions. Programming state machines that sprinkle state changes across functions can be a source of implementation faults. There have been several examples of faulty state transitions discussed in the literature [9] and TinyOS mailing lists. Note that the fault we discussed in Section 8.1 is also an instance of this type. We note that it is easy to see that state transitions are captured by the control-flow and hence, faulty transitions can be identified from a control-flow trace easily. We omit detailed case studies in the interest of space. We note that the state-of-the-art techniques such as deriving finite state machines [9] can detect most of these faults at compile-time itself. However, the faults such as in Section 8.1 could not have been detected with [9] as the state itself was missing.

9 Performance Evaluation

In this section, we show that our tool incurs low run-time and compile-time overheads and the trace sizes are viable. The run-time overhead includes CPU cycles required for executing the instrumented instructions, writing to the external flash storage and the amount of external storage or trace size. We measure the energy consumed by CPU and external flash to estimate the run-time overhead. We first discuss our benchmark suite and then present the run-time overhead in terms of energy consumption and trace size. Next, we show our trace size compares favorably with the state-of-the-art technique. Finally, we present the compile-time overhead in terms of code size and RAM used by our tool.

9.1 Evaluation Benchmarks and Setup

Since there is no competing tracing solution available for nesC programs (see Section 11) and benchmarks lack, we chose four characteristic TinyOS applications that are widely studied in other works [3] namely, Blink, Oscilloscope, Surge and CntToLedsAndRFM as well as a large TinyOS application called LRX. LRX is a reliable large data transfer module developed as part of the Golden Gate Bridge monitoring project for our evaluation. We used the SingleHopTest to drive the LRX module.

These benchmarks are included with TinyOS 1.x. The Blink application toggles the red LED every second. The Oscilloscope application records light sensor values every 125 milliseconds and sends the accumulated values as a message to the base station periodically. Oscilloscope can generate large amounts of trace information because of high frequency timer, thus stretching our technique. The Surge application is similar in functionality to Oscilloscope but uses a slower timer that ticks about every 2 seconds but sends message every time it senses. CntToLedsAndRFM broadcasts as well as displays the count every 250 milliseconds. LRX source contains about 1350 lines of nesC code making it one of the largest TinyOS application in the contributions directory. LRX benchmark stress-tests our implementation as it has complex control-flow.

9.2 Energy Consumption

We used Atemu, a cycle-accurate emulator to record the energy consumed per hour for each of the benchmarks in our suite. We ran every benchmark for 1 hour on Atemu and measured the CPU, flash, and total energy usage. We verified the results for a few benchmarks from the simulator on our mica2 motes testbed as direct energy measurement for all our benchmarks is difficult and human error prone.

For each benchmark, we measured the overhead of both inter-procedural and intra-procedural tracing. The inter-procedural case is referred to as *inter*. For the intra-procedural case, we noted that functions with trivial control-flow need not be recorded if the callers of those functions are traced. So, we split intra-procedural tracing into two types, naive or *intra-all*, which records all traced functions, and *intra*, that applies the above optimization. Observe that *intra-all* can be thought of as function call tracer since function call tracing need to trace all functions as it does not exploit the control-flow. NodeMD [11] proposes function call tracing and can be treated as *intra-all*.

Since the overhead of tracing depends on the components included in the tracing, we traced the important nesC components including the application component (e.g., `SurgeM` for Surge) as well as the system components such as LEDs (e.g., `LedsC`), sensor (e.g., `PhotoTempM`), single-hop network layer (e.g., `AMStandard`) or multi-hop network layer (e.g., `MultihopEngineM`), and timer (e.g., `TimerM`). The order in which the components were included in the simulation is application, LED, sensor, network layer, and timer.

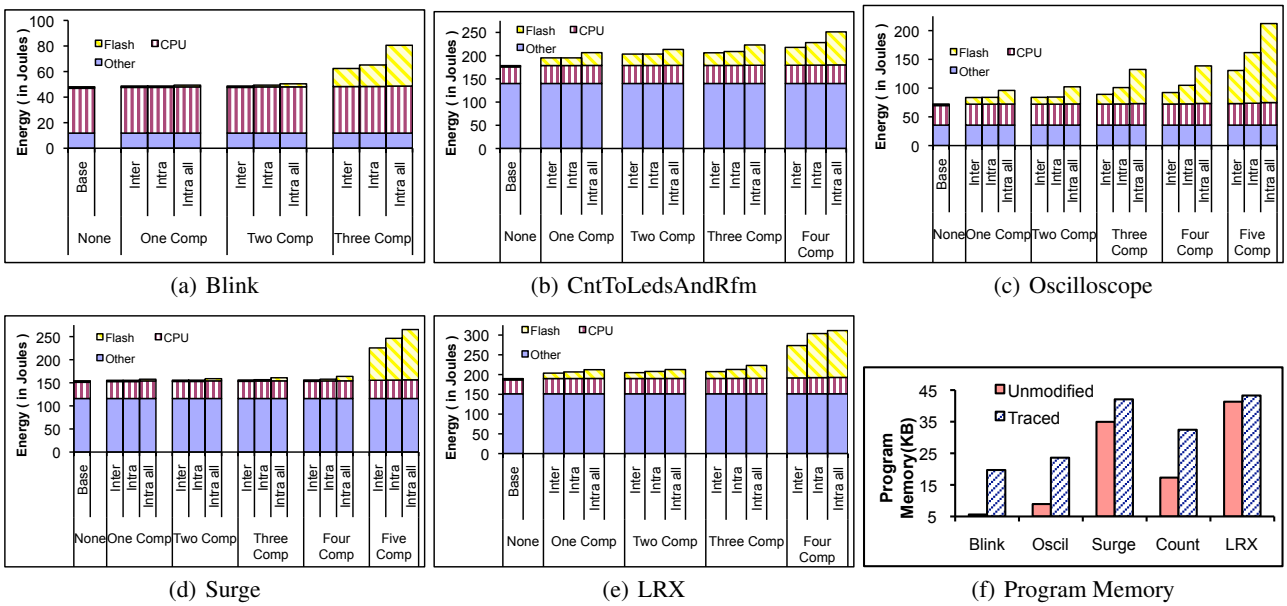
The results of our simulation are shown in Figure 10. The energy consumed is divided into three parts: CPU, shown as red vertical stripes, flash, shown as yellow slanting lines and the other (such as network layer, LEDs), shown as blue filled bars. The baseline case with no tracing is shown as the first bar followed by energy consumed by a progressively increasing number of traced components. For each of the traced components category, the order of the three bars is *inter*, *intra* and *intra-all*.

First, we observe that the flash overhead dominates the CPU overhead in general as we add more components. Next, we observe that *inter* clearly consumes much lower overhead compared to both the *intra* and *intra-all* as less as 378% (for Surge, 4comp). This is because fewer bytes are recorded in the case of *inter*, and flash energy dominates. It's interesting to note that the *intra-all* (akin to naive function call tracing) uses 39.9% (for Count, 2comp) to 378.4% (Surge, 4comp) more energy than *inter*. Further, we observe that *intra* performs much better than *intra-all* because of the presence of large number of simple wrapper functions in nesC, which do not have control-flow. Compared to *inter*, *intra* uses 0 % to 101.4% (for Blink, 2comp) more energy.

We observe that the inter-procedural tracing overhead is low (0.88% for Surge to 9.43% for count) when only application component is traced. As we start tracing more system components, the overhead increases. However, it is interesting to note that including even most system components (LEDs, sensors, and network layer) except Timer did not increase the overhead (1.31% for Surge to 29.45% for Oscilloscope) that much. This is true even for LRX which has about 1350 lines of code. It is also interesting to note that tracing overhead directly depends on how often the node is awake in addition to the components traced. Hence, Oscilloscope has more overhead than LRX or Surge, which are mostly asleep. For real-world sensing applications that are awake only once in a few minutes, our tracing would generate low overhead. When Timer is included, the overhead increases noticably and ranges from (26.78% for Count, 4comp to 81.76% for Oscilloscope, 5comp). This is due to the presence of large loops. From the results, we infer that control-flow tracing indeed incurs low overhead for tracing most of the application and system components.

9.3 Trace Compression

We show that our compression scheme is effective and generates smaller traces. We recorded the trace using Atemu, which logs the flash in a file at the end of the simulation. We parse the trace to uncompress it at the end of simulation. The data compression ratio is the ratio of uncompressed size to compressed size. The trace sizes and compression ratio recorded for each of the benchmarks are shown in the Ta-



(a) Blink (b) CntToLedsAndRfm (c) Oscilloscope

(d) Surge (e) LRX (f) Program Memory

Figure 10. Run-time overhead for control-flow tracing measured as total energy shown in three parts, namely, CPU, flash and all others. Figure 10(f) shows program memory size comparison of unmodified and instrumented version of benchmark programs. The y-axis units is kilobytes(KB).

ble 1. We observe that the amount of trace information generated in 30 minutes for a normal WSN application is less than a KB (568 for Surge) and for a high-throughput applications is about a few KBs (4864 for Oscilloscope). For a complex application like LRX that contain multiple loops and several function calls, the trace size is still manageable and can further be reduced by tracing only important parts of the application.

Table 1. Trace size for 30 minutes run.

Benchmark	Compressed trace size (B)	Compression Ratio
Blink	344	21.71
Oscilloscope	4864	11.99
Surge	568	14.11
Count	3664	4.625
LRX	93200	1.74

An alternative to our scheme is to capture the control-flow using a simple instruction-level tracing scheme, which records the program counter of every instruction executed can be used. We assume that the program counter can be encoded in a single byte, even though it would take more. We used Avrora, a cycle-accurate simulator, to record the actual number of instructions executed over a period of 30 minutes and this instruction-level trace size ranges from 2.12 MB (Blink) to 421.6 MB (Count) for our benchmarks. The instruction-level scheme captures all concurrent event interleavings exactly, but the size of the generated trace is prohibitively large. The uncompressed trace generated by our technique is much smaller than the instruction-level tracing.

9.4 Trace Size Comparison

In this section, we compare the size of the trace generated by our algorithm with the MR algorithm and show that our algorithm is comparable in size or better.

We, first, observe that the MR algorithm has to be adapted to the concurrent environment similar to our algorithm, i.e., applying it to every execution unit (task/event). Since there is no open implementation available of the MR algorithm (there are no experimental results in the original paper [16]), we estimate the trace size generated by the MR algorithm using the trace generated by our algorithm.

The two algorithms differ in the way they handle non top-level functions (functions other than tasks or events) containing loops. Both algorithms record the loop paths for every iteration of a loop. In the MR algorithm, each loop path gets a globally unique path identifier, thus increasing the global identifier space significantly. However, in our algorithm, we use locally unique identifiers for loop paths and use the function context to distinguish the loop paths across functions. Since we record more entries than the MR algorithm for non top-level functions with loops but use fewer bits per entry in the trace, the actual trace size would depend on the number of iterations of the loops inside non top-level functions.

Let n be the number of entries in the trace generated by our algorithm. Let f be the number of non-top level functions containing loops. Let o and t be the number of bits used to encode an entry in the trace by our algorithm and the MR algorithm respectively. Note that, $t > o$, o is 16 bits in our case. Our trace size is $n \times o$ whereas the MR trace size is $(n - 2 \times f) \times t$. The two entries created for the start and path identifier of any non top-level function containing loops will not exist in the trace generated by the MR algorithm and for this reason, we subtract $2 \times f$ from the number of entries.

We obtained the values of n and f from the uncompressed trace for each of the benchmarks. The results are shown in Table 2. The columns in the table correspond to the size of the trace generated by the MR algorithm normalized by size of the trace generated by our algorithm. Since the value of t depends on the implementation, we vary t from 17 bits

to 32 bits, which is 1 bit to 2 bytes more than o. We observe that the trace size generated by the two algorithms are comparable even when the global identifier uses just 1 bit more and this is because there are only few non top-level functions with loops in the benchmark programs. When the global identifier uses one byte or 2 bytes more, our algorithm saves considerable space (14% to 83%). Therefore, we conclude that in WSN settings, our algorithm, which uses less CPU cycles per instrumentation, simpler to implement and saves trace size, is preferable over a direct adaptation of the MR algorithm.

Table 2. Trace size comparison with the MR algorithm

Benchmark	t = 32 bits	t = 24 bits	t = 20 bits	t = 17 bits
Blink	1.526	1.145	0.954	0.811
Oscilloscope	1.809	1.357	1.131	0.961
Surge	1.82	1.367	1.140	0.968
Count	1.83	1.374	1.145	0.973
LRX	1.79	1.345	1.121	0.952

9.5 Memory Consumption

We discuss the program and data memory requirements of our tracing scheme for our benchmarks. The program memory represents the code size and the data memory represents the working memory/RAM size. We obtained these values by compiling with default switches; that is, OS code-optimization was turned on. We note that there are two components to program memory overhead: *fixed* and *variable*. The former component is due to the internal algorithms such as the generic compression used by our tracing component and the device driver code for flash component. The latter component corresponds to instrumentation. The program memory used by unmodified and traced versions are shown in Figure 10(f). For simple programs like Blink and Oscilloscope, the overhead is high because of the fixed components. But for large programs such as Surge and LRX, the fixed component increase is reduced because some device drivers are already included as part of the program. Another reason for smaller overhead in Surge and LRX is that the compiler removes inlining of function calls in the traced version. The effect of removal of inlining is not high as our energy overhead results indicate. The increase in program size is modest and for most programs well under the limits of 128 KB for mica (mica2, micaz) family or 48 KB for telosb families of motes. Since the nesC compiler aggressively inlines the program, reducing the inlining can help large programs.

The data memory requirement is around 950 bytes and is usually independent of the instrumented program. The major contributor to the data memory are the internal buffers (640 bytes) used to store the trace for future compression and for recording into flash while allowing the application to proceed. The internal buffer size is configurable and here we show the highest configuration used for LRX. For most programs, half that size is enough. The split of the data memory is shown in Table 3.

Table 3. Division of data memory overhead in bytes

Datastructure	Memory size in bytes
Flash Pages(16)	256
Circular Buffer(2)	384
Miscellaneous	300

10 Limitations

Our approach is a first step to creating a post-mortem replay debugger for WSNs. As with any tracing mechanism, manual labor is involved in analyzing traces. We are working on automating the trace collection from individual sensors.

By capturing control-flow, our approach is generic enough to diagnose a large class of errors that change the control-flow including logical errors, high-level race conditions, node reboots, network failures (node/link failures), and memory errors with varying overhead. However, specific approaches can be less expensive for the respective targeted classes of errors such as SafeTinyOS [3] for memory-specific errors, or can require less manual intervention such as Sympathy [19] or PAD [13] for network failures. There are cases where control-flow tracing cannot distinguish the abnormal from normal such as malicious entities corrupting function return addresses in the stack or entries of the routing table. These security errors are important and can be mitigated using authorization and authentication mechanisms.

Since our control-flow tracing is limited to the granularity of basic blocks, reconstructing exact instruction-level execution is not directly possible. If multiple instructions inside a basic block could race with an interrupt, any of the interleavings between this basic block and the interrupt would produce exactly the same control-flow trace. In the post-mortem analysis, the developer would not be able to localize the fault with the control-flow trace alone. We note that during post-mortem analysis, tools such as Chess [17] that try all possible interleavings of the interrupt with the basic block instructions accessing shared data can be helpful. The combination of our tracing technique with such an analysis is a potential avenue of future research. The static data race detection in nesC prevents low-level race conditions.

As noted earlier, the trace size can increase when many components are traced. It is usually the case that the developer wants to trace only those application components that are more likely to contain logical errors rather than well-tested system components.

11 Related Work

Existing work in debugging for WSNs can be classified into (1) tools that aid in automating debugging, (2) tools that provide insight into the network, and (3) tools that analyze programs for testing or extracting higher-level abstractions.

11.1 Automated Debugging

Ramanathan et al. propose Sympathy [19], which collects network metrics such as connectivity and data flow, and node metrics periodically for failure detection. The source of a failure can be narrowed down to a node or sink or the communication path itself. Liu et al. propose PAD [13], which uses lightweight network monitoring and bayesian network-based analysis to infer network failures and their causes. Sympathy and PAD focus on network faults and do not handle application implementation faults. Krunic et al. [11] propose NodeMD, a tool that can detect stack overflows, live-locks, and deadlocks. NodeMD is similar to our work in that it instruments the code and encodes high-level events with small numbers of bits to record traces. However, a recorded trace does not contain the exact control-flow path informa-

tion and hence, cannot be automatically replayed later to reproduce faults. Moreover, the tracing is not designed for event-based concurrent systems like ours. Shea et al. [20] propose an optimized way for storing function call traces similar to NodeMD but it lacks inter-procedural control-flow tracing. Luo et al. [15] propose to record all events in a given time period and replay them within the node at later point in time. This solution is mainly proposed for in-field post-deployment testing where trace sizes can be controlled rather than for a post-deployment debugging tool. Herbert et al. [6] present an invariant-based application-level run-time monitoring tool that can be used to detect errors in WSN applications. This tool does not have the diagnostic framework required to find the root cause of the error detected. Cao et al. [2] propose Declarative TracePoints, which provides a SQL-based language interface for debugging. Khan et al. [8] propose Dustminer, a machine learning-based approach to diagnosis. Dustminer uses the pattern mining algorithm to distinguish good executions from bad executions in the logs collected.

11.2 Visibility

Since motes do not include a user-friendly interface but merely three LEDs, it is important to provide insights into the state of the network to gain confidence that it is functioning properly. Whitehouse et al. [22] propose Marionette, a tool suite that supports function calls and memory reads ("peeks") and writes ("pokes") on a remote node. This allows a programmer to check the state of the network and alter it if needed. Yang et al. [23] propose Clairvoyant, which is a remote debugging tool that provides a `gdb` like debugging and allows a programmer at the base station to control the execution of the nodes in the network or to inspect the state of the network. Kothari et al. [10] propose Hermes, a lightweight framework and tool that provides fine-grained and dynamic visibility without modification to end applications. While these tools provide an elegant way of inspecting state of a WSN, we provide a complementary method in a way like `printf` based debugging is complementary to `gdb` debugging.

11.3 Tools Based on Program Analysis

Nguyen and Soffa [18] present an application graph abstraction to represent nesC programs. Kothari et al. [9] use symbolic execution to extract the finite state machine implied by the TinyOS code, which aids in better understanding of the code. While these tools are helpful in development and testing, they are not meant for run-time debugging.

12 Conclusions and Future Work

In this paper, we have shown that program tracing can be performed efficiently and accurately in wireless sensor networks. To that end, we have introduced a novel trace encoding scheme and a basic trace compression scheme. We have applied our scheme to several case studies and applications.

Backed by this evidence of the viability of our approach, we are exploring different improvements to our technique. For instance, the unrolling of loops can potentially reduce the number of entries into trace buffers. Furthermore, we are working on modular computation of inter-procedural paths which involve function calls with loops.

13 Acknowledgement

We thank the anonymous reviewers and Kamin Whitehouse for their insightful comments. This research is supported, in part, by the National Science Foundation (NSF) under grant 0834529. Any opinions, findings, conclusions, or recommendations in this paper are those of the authors and do not necessarily reflect the views of NSF.

14 References

[1] T. Ball and J. R. Larus. Efficient path profiling. *Micro '96*.

[2] Q. Cao, T. Abdelzaher, J. Stankovic, K. Whitehouse, and L. Luo. Declarative tracepoints: A programmable and application independent debugging system for wireless sensor networks. *SenSys '08*.

[3] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient memory safety for TinyOS. *SenSys '07*.

[4] D. Geels, G. Altekar, P. Maniatis, and T. Roscoe. Friday: Global comprehension for distributed replay. *NSDI '07*.

[5] W. B. Heinzelman, A. P. Chandrakasan, and H. Balakrishnan. An application-specific protocol architecture for wireless microsensor networks. *IEEE TWC*, 1(4):660–670, 2002.

[6] D. Herbert, V. Sundaram, Y. H. Lu, S. Bagchi, and Z. Li. Adaptive correctness monitoring for wireless sensor networks using hierarchical distributed run-time invariant checking. *ACM TAAS*, 2(3):8, 2007.

[7] M. Keller, J. Beutel, A. Meier, R. Lim, and L. Thiele. Learning from sensor network data. *SenSys '09*.

[8] M. Khan, H. Le, H. Ahmadi, T. Abdelzaher, and J. Han. Dustminer: Troubleshooting interactive complexity bugs in sensor networks. *SenSys '08*.

[9] N. Kothari, T. Millstein, and R. Govindan. Deriving state machines from TinyOS programs using symbolic execution. *IPSN '08*.

[10] N. Kothari, K. Nagaraja, V. Raghunathan, F. Sultan, and S. Chakradhar. Hermes: A software architecture for visibility and control in wireless sensor network deployments. *IPSN '08*.

[11] V. Krunic, E. Trumpler, and R. Han. NodeMD: Diagnosing node-level faults in remote wireless sensor systems. *MobiSys '07*.

[12] J. Larus. Whole program paths. *PLDI '99*.

[13] K. Liu, M. Li, Y. Liu, M. Li, Z. Guo, and F. Hong. Pad: Passive diagnosis for wireless sensor networks. *SenSys '08*.

[14] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang. D3s: Debugging deployed distributed systems. *NSDI '09*.

[15] L. Luo, T. He, G. Zhou, L. Gu, T. F. Abdelzaher, and J. A. Stankovic. Achieving repeatability of asynchronous events in wireless sensor networks with envirolog. *INFOCOM '06*.

[16] D. Melski and T. W. Reps. Interprocedural path profiling. *CC '99*

[17] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. *OSDI '08*

[18] N. T. M. Nguyen and M. L. Soffa. Program representations for testing wireless sensor network applications. *DOSTA '07*.

[19] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. *SenSys '05*.

[20] R. Shea, M. Srivastava, and Y. Cho. Scoped identifiers for efficient bit aligned logging *DATE '10*

[21] T. Sookoor, T. Hnat, P. Hooimeijer, W. Weimer, and K. Whitehouse. Macrodebugging: global views of distributed program execution. *SenSys '09*.

[22] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: using RPC for interactive development and debugging of wireless embedded networks. *IPSN '06*.

[23] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. *SenSys '07*.