Diagnostic Tracing for Wireless Sensor Networks

VINAITHEERTHAN SUNDARAM, PATRICK EUGSTER, XIANGYU ZHANG, and VAMSIDHAR ADDANKI, Purdue University

Wireless sensor networks are typically deployed in harsh environments, thus post-deployment failures are not infrequent. An execution trace containing events in their order of execution could play a crucial role in postmortem diagnosis of these failures. Obtaining such a trace however is challenging due to stringent resource constraints. We propose an efficient approach to intraprocedural and interprocedural control-flow tracing that generates traces of all interleaving concurrent events and of the control-flow paths taken inside those events. We demonstrate the effectiveness of our approach with the help of case studies and illustrate its low overhead through measurements and simulations.

Categories and Subject Descriptors: D.2.5 [**Software Engineering**]: Testing and Debugging—*Debugging-aids, Tracing*

General Terms: Design, Experimentation, Reliability

Additional Key Words and Phrases: Embedded debugging, tracing, wireless sensor networks, diagnosis

ACM Reference Format:
Sundaram, V., Eugster, P., Zhang, X., and Addanki, V. 2013. Diagnostic tracing for wireless sensor networks. ACM Trans. Sensor Netw. 9, 4, Article 38 (July 2013), 41 pages.
DOI: http://dx.doi.org/10.1145/2489253.2489255

1. INTRODUCTION

Wireless sensor networks (WSNs) are being increasingly deployed to monitor physical phenomena in various scientific, military, and industrial domains. As WSNs are deployed in austere environments, such as glaciers and volcanoes, post-deployment failures are not uncommon. Post-deployment failures have been observed in most of the research deployments, and often such failures went unexplained for long periods [Beutel et al. 2009]. Unconventional programming models and lack of kernel support make WSNs prone to complex faults, such as race conditions, which are triggered by unexpected interleaving of events in the real world. Diagnosing post-deployment failures thus constitutes an important problem, but the inherent stringent resource constraints in WSNs make such diagnosis very challenging.

There have been several debugging solutions proposed for WSNs. Automated debugging tools [Ramanathan et al. 2005; Liu et al. 2008; Krunic et al. 2007; Luo et al. 2006; Herbert et al. 2007] mostly monitor the network and thus do not provide much

---

This article is an extended version of Sundaram et al. [2010], which appeared in *Proceedings of he 8th ACM Conference on Embedded Networked Systems (SenSys'10)*.
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 necessaily reflet the views of NSF.
Authors' addresses: V. Sundaram and V. Addanki, School of Electrical and Computer Engineering, Purdue University; P. Eugster and X. Zhang, Department of Computer Science, Purdue University; corresponding author's email: vsundar@purdue.edu.

DOI: http://dx.doi.org/10.1145/2489253.2489255

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 runtime monitoring and debugging. Remote control approaches [Whitehouse et al. 2006; Yang et al. 2007] provide insight into, and control of, remote sensor nodes. These approaches incur a substantial overhead and require the developer to navigate through the program at runtime. Finally, program analysis-based tools [Nguyen and Soffa 2007; Kothari et al. 2008] provide higher-level understanding of programs but fail to quickly pinpoint the exact causes of faults or to exhibit high complexity.

Replay debugging is a powerful runtime debugging technique for diagnosing complex faults and is especially useful for debugging distributed applications because of the inherent nondeterminism [Geels et al. 2007; Liu et al. 2008; Sookoor et al. 2009]. Replaying requires obtaining the trace of the ordered sequence of events or control flow path taken and the input values. This may not present insurmountable issues in a wired setting; obtaining it in a WSN, however, can be prohibitively expensive in terms of bandwidth required to transfer it from a node to the base station. Tight constraints 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, therefore, is to decide what information to record that allows partial but faithful WSN replay debugging while satisfying resource constraints.

1.1. Control-Flow Tracing

For WSN settings, we argue for recording control flow information. On one hand, this information is effective in fault diagnosis, as many faults manifest through abnormal control flow. Knowledge of the control flow can help a great deal in diagnosis for a large class of errors that change the control flow, including logical errors, high-level race conditions, node reboots, network failures (i.e., node/link failures), and memory faults. Despite sophisticated techniques and tools just briefly summarized, one of the commonly used flavors of debugging is LED debugging [Shea et al. 2010], where the developer manually annotates the program to switch LEDs on and off in order to notify the success or failure of events of interest; one of the critical pieces of information the developer is trying to get by this type of debugging is the control flow path taken.

On the other hand, 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. Therefore, if the control flow path is suitably encoded, the repetitive sequences of actions can be highly compressed, leading to resource-efficient recording of control-flow information.

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 message content is reflected in the control flow. Likewise, if the sensed value is beyond a threshold, the action taken by the sensor is captured in the control flow. There are faults, such as routing table corruption by malicious entities, where recording input values could be useful. However, such an approach becomes impractical for WSN settings due to the large trace sizes and stringent constraints on memory resources. Control-flow traces yield a good compromise that allows partial replay of the execution.

1.2. Real-World Application

Consider, as an example, the PermaSense project [Hasler et al. 2008], which strives for collecting geophysical data with WSNs in the harsh environment of the Swiss Alps. The deployment had severe performance degradation after six months of deployment

(March 2009). Extensive resets of nodes, up to 40 resets per node per day, were observed for three months [Keller et al. 2009].

The cause of the bug was a lookup task whose running time increased with the data stored in the external flash memory. Since external flash was used to store sensor data, after several months of deployment, that task's execution time became large enough to starve other tasks from executing, resulting in task queue overruns. A safety mechanism to survive task queue overruns in the field was a soft reset, which reinitialized memory and restarted the application. However, the fault survived soft resets and continued to cause more resets, because the task's execution time depended on the external flash, which was left unmodified by soft resets.

The diagnosis took months and several expensive trips to the mountain top. The developers obtained RAM dumps from the deployed nodes and used them to reproduce the failure in a controlled setting.

If 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 have contained the lookup task and the repeated interaction with the external flash device. This would have revealed the problem instantly and saved expensive trips to the mountain top.

1.3. Our Contributions

In this article, we propose a novel efficient tracing technique that encodes and records the interprocedural control flow of all interleaving concurrent events. Our technique has a low footprint, which allows it to deal with long-running real-life programs and is achieved by statically analyzing the program and instrumenting only a few statements inside event handlers. At runtime, the trace is recorded in memory and compressed before being committed to nonvolatile 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. To localize the fault, the trace received can be analyzed by human or automated tools, or it can be replayed in a controlled environment, such as a trace simulator. While explained and implemented in the context of TinyOS [Levis et al. 2005], that is, in TinyOS 1.x and TinyOS 2.x, our techniques are generic.

In summary, the main contributions of our article are as follows.

(1) We present a *coarse-grained* tracing technique for nesC programs to trace at the level of execution units, that is, tasks and events.
(2) We introduce a *fine-grained* tracing technique for that is, nesC programs in steps. First, an intraprocedural control flow tracing technique is proposed to facilitate locating the point at which execution transitions from one execution unit to another. The scheme precisely captures the effect of event interleavings on control flow by faithfully recording the fine-grained control flow paths. Second, an interprocedural control flow tracing technique based on a *novel static program analysis* is presented to deal with nested function calls. Third, we present an algorithm to trace a subset of functions in event-driven concurrent programs, such as TinyOS applications.
(3) We present the tool called *TinyTracer*, an implementation of our tracing design for TinyOS/nesC programs. TinyTracer is easily adaptable to other OSes such as Contiki [Dunkels et al. 2004], as it analyzes C programs. TinyTracer also features an effective custom-tailored compression technique.
(4) We illustrate the effectiveness of our approach through case studies of common types of node-level faults in TinyOS, including a case study of a previously unknown bug in TinyOS 1.x that remained undetected for many years.
(5) We demonstrate the efficiency of our solution through performance measurements on several typical WSN applications as well as one of the largest nesC programs

which was used in Golden Gate Bridge monitoring. We present a detailed comparison with three state-of-the-art techniques [Shea et al. 2010; Krunic et al. 2007; Melski and Reps 1999]. We show our approach generates smaller traces (up to 87% smaller) while incurring lower energy overhead (up to 79% lower) compared to the state of the art.

1.4. Roadmap

The rest of the article is structured as follows. Sections 2 and 3 present our efficient tracing design, including our coarse-grained and fine-grained tracing techniques. Section 4 discusses implementation challenges and introduces our tool TinyTracer. Sections 5 and 6 present the evaluation of our approach, and Section 7 discusses its limitations. Section 8 contrasts our approach with related work. Section 9 concludes the article.

2. TASK- AND EVENT-LEVEL TRACING

We first focus on the coarse-grained tracing of thread interleavings but not on the detailed execution paths inside threads. Information on interleavings is essential to diagnosing many complex failures of TinyOS applications which are due to unexpected interaction of different interrupt handlers. Simultaneous modification of a receiver buffer by an application and the network layer is a common example of such a fault [Khan et al. 2008].

2.1. TinyOS Execution Model

The TinyOS event-driven execution model poses unique programming challenges but allows for efficient solutions for severely resource-constrained environments compared to the execution model of general-purpose program environments, which allow for arbitrary thread interleavings. We summarize the TinyOS execution model and present a key observation about it that enables efficient tracing.

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 other components 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 for real-time constraints. More particularly, low-priority operations should give way to critical operations, such as interrupts from hardware. There are two sources of concurrency in TinyOS: *tasks* and *interrupt handlers* (or *async events*, also simply called *events* in this article). Tasks and events are normal functions with special annotations. Tasks are a deferred computation mechanism. In the absence of events, 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.

The operation to get a value from a sensor is often split into multiple phases. First, a command call (i.e., down-call) is made to a service component to start the operation,

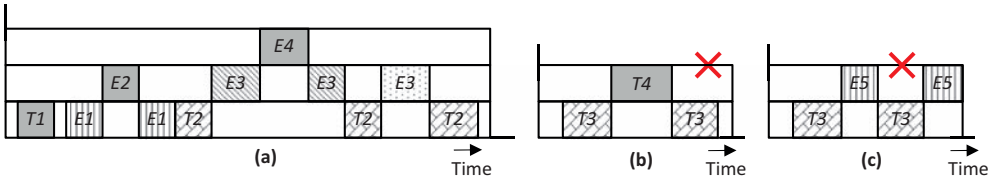


Fig. 1. TinyOS Model. Boxes depict execution of tasks or events. The y-axis represents the preemption level. Tasks and events are labeled with *T*'s and *E*'s, respectively. (b) and (c) show examples of illegal TinyOS executions.

which is carried out by multiple tasks. Later, when the operation is completed, an interrupt occurs, generating a callback (i.e., event/up-call) to the consumer component, which can process the result in the handler immediately or post tasks to handle later. 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 sequential or nested. This enables efficient tracing as shown in Section 2.3. In contrast, a traditional thread-based concurrency model allows arbitrary thread interleaving.

To illustrate the TinyOS concurrency model, consider Figure 1, which shows a snapshot of how tasks and events execute in the TinyOS concurrency model. Boxes depict the execution of tasks or events. Tasks and events are labeled with T's and E's, respectively. Boxes with the same pattern in Figure 1 represent the same execution instance of a task or an event, and their labels denote the names of tasks or events. Observe that a task or an event may be preempted, giving rise to multiple boxes with the same pattern. 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 received by the handler *E3*. Once *E3* completes, *T2* again resumes and runs to completion without further preemption.

Figures 1(b) and 1(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.

2.2. Challenges

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, WSNs 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 cannot tell that the first two *E3*'s belong to the same instance and that the third *E3* is a stand-alone one, whereas the three *T2* blocks belong to the same instance.

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

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

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*.

2.3. Approach

The nesting structure of the TinyOS application execution can be exploited to describe traces 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 are tasks or events. 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(a) can be depicted by the following string of grammar. Overbraces represent the nestings.

$$\overbrace{\overbrace{T1\ \textbf{\textit{end}}}\ E1\ \overbrace{\overbrace{E2\ \textbf{\textit{end}}}\ \textbf{\textit{end}}}^{}}^{}\ T2\ \overbrace{E3\ \overbrace{\overbrace{E4\ \textbf{\textit{end}}}\ \textbf{\textit{end}}}^{}\ \overbrace{E3\ \textbf{\textit{end}}}^{}\ \textbf{\textit{end}}}^{}\ \textbf{\textit{end}}.$$

The encoding not only captures the nesting, which allows for 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 or event identifiers.

2.4. Grammar Parsing

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, for example, the second and third *T2* blocks can be reproduced during parsing by inspecting the stack's state.

The parse algorithm is presented in Algorithm 1. It is a top-down recursive parser. Each method corresponds to a rule in Figure 2. The method body is constructed from the right-hand side of the rule. Recursions in the grammar rule are reflected as recursive calls in the algorithm. The main feature of the algorithm is the use of a parse stack. The stack state discloses the current nesting of an event. For example, for the trace presented earlier, PUSH (*T1*) is performed at the first trace entry; a POP () is performed upon the second entry, revealing the termination of *T1*. Identifiers *E1* and *E2* are pushed due to the third and fourth entries and then popped in a last in, first out (LIFO) fashion, denoting that the preempting execution *E2* terminates before *E1* does. Note that, after parsing "*E3 E4* ***end end***," identifier *T2* is on top of the stack. Therefore, the following *E3* entry indicates that the current *T2* task is preempted by a different instance of *E3*. Moreover, one can infer that some instructions of *T2* must have been executed between *E3* and the preceding ***end*** although an explicit *T2* block is not present in the trace.

ALGORITHM 1: A Top-Down Recursive Parsing Algorithm for Parsing Coarse-Grained Trace. Assume stack and stack procedures PUSH () and POP (). Procedure NEXTTOKEN () is used to retrieve next entry from the trace. Procedure ENDOFTRACE () returns true if the end of the trace is reached. Procedures ISEVENT () and ISTASK () return true if the given ID is an event or task. ASSERT () aborts execution if the predicate is false.

```
procedure PARSETRACE ()
  while ¬ ENDOFTRACE () do
    PARSEEUNIT ()
  end while

procedure PARSEEUNIT ()
  id ← NEXTTOKEN ()
  ASSERT (ISEVENT (id) or ISTASK (id))
  PUSH (id)
  while NEXTTOKEN ()≠ end do
    PARSEEVENT ()
  end while
  POP ()
  ASSERT (NEXTTOKEN ()= end)

procedure PARSEEVENT ()
  id ← PARSEID ()
  ASSERT (ISEVENT (id))
  PUSH (id)
  while NEXTTOKEN ()≠ end do
    PARSEEVENT ()
  end while
  POP ()
  ASSERT (NEXTTOKEN ()= end)
```

3. 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 for computing interprocedural paths, which reduces the trace size and CPU cycles used compared to the state-of-the-art techniques.

3.1. Challenges

The task- and event-level traces generated in Section 2 capture high-level interleavings. However, they are insufficient because they do not have fine-grained control flow information within the tasks or events and also do not contain the exact preemption points. Knowledge of fine-grained preemption information is useful for diagnosis, because different preemption points can lead to different program states if the interleaved executions access shared variables. As reported in Khan et al. [2008], a large portion of TinyOS application faults fall into this category of incorrect shared memory access. One such fault is discussed in Section 5.3.1.

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 switche, 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. Furthermore, if a loop execution were preempted, the loop iteration number cannot be recovered from the program counter alone. Another possible solution is to instrument TinyOS applications so that the program counter of each executed instruction could be recorded. In such a trace, an unexpected program counter alternation indicates preemption. Clearly, such a solution is very expensive, as nontrivial instrumentation has to be inserted for each instruction.

3.2. Approach

We propose a technique for tracing inside tasks and events and recording 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 interleavings induce different conditional branches being taken, the control flow trace precisely captures the effect of interleavings by retaining the branches taken. The challenge of our design thus lies in efficiently representing control flow paths.

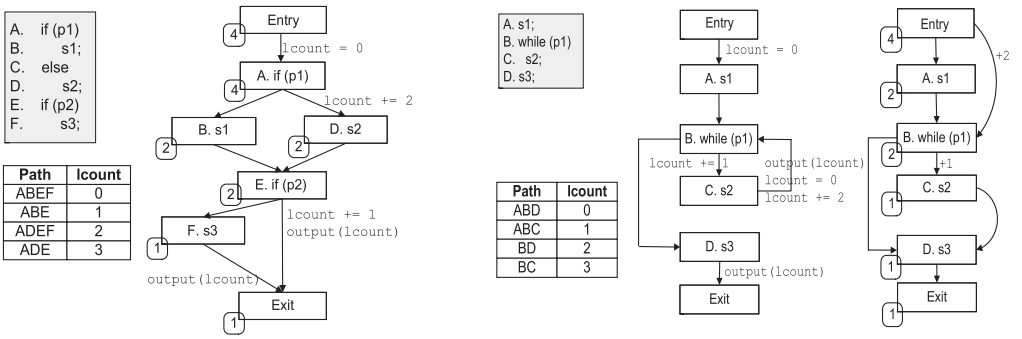
3.3. Intraprocedural Control Flow Tracing

For presentation clarity, 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 intraprocedural control flow paths. More specifically, n acyclic intraprocedural control flow paths of a given procedure can be represented with an integer from 0 to $n-1$ [Ball and Larus 1996].

3.3.1. Background. In their seminal paper, Ball and Larus [1996] proposed an efficient algorithm (referred to as the BL algorithm) for computing the optimally-encoded intraprocedural control flow path identifier taken during execution. The BL algorithm features translating acyclic path encodings to instrumentations on control flow edges. At runtime, 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(a). 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 node A, `lcount` is incremented by 2. If the false branch is taken at node E, `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.

ALGORITHM 2: 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 "`output(lcount)`"



(a) Simple. (b) Loop.

Fig. 3. Examples for Ball-Larus path encoding. Instrumentations are marked on control flow edges. Node annotations (i.e., boxes with rounded corners) represent the number of paths leading to *Exit*. (a) An example of encoding simple control flow paths; (b) an example of encoding control flow paths with loops. The example code and its path encoding are shown on the left side of both figures. In (b), the left part shows the control flow graph of the program with its instrumentation, and the right part shows the transformed acyclic graph obtained by replacing the back edge with two dummy edges. The transformed acyclic graph is used to compute the instrumentation. Variable `lcount` holds the path identifier, which is emitted at back-edge and procedure exit.

The basic algorithm is presented in Algorithm 2. 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(a), node A has the annotation (rounded box at the corner) of `4`, which is the sum of the annotations of nodes B and D, meaning there are four paths leading from node A to *Exit*. The instrumentation on an edge $n \rightarrow m$ is computed as increasing the path identifier 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 $A \rightarrow D$ is `lcount+=2`, as the annotation of A's preceding child B is `2`, meaning the range `lcount` $\in$ [0,2) is allocated to the paths following the edge $A \rightarrow B$ and `lcount` $\in$ [2,4) is allocated to the paths following $A \rightarrow D$. Finally, the algorithm instruments the end of a function by emitting the path identifier to the trace.

Loop paths refer to the control flow paths that end at the loop exit nodes, which are the nodes with back edges in a CFG. 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 by introducing dummy edges from *Entry* to the loop head and loop exit nodes to *Exit*. Figure 3(b) 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 identifier increments—the program itself is not transformed. The path identifier increments are computed in the same way as previously. 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 identifier to the trace, then resets the counter, and finally increments the counter by two, due to the increment on the dummy edge from *Entry* to B. One can observe that these instrumentations generate the encodings of acyclic paths on the left. For example, the trace of *Entry A B C B C B D Exit* is encoded as *1 3 2*. The detailed algorithm can be found in Ball and Larus [1996]. Note that the

$$Trace \rightarrow EUnit* \qquad EUnit \rightarrow Id\ (Event|Path)*\ Path\ \textbf{\textit{end}} \mid \epsilon$$
$$Id \rightarrow \textbf{\textit{Tid}} \mid \textbf{\textit{Eid}} \qquad Event \rightarrow \textbf{\textit{Eid}}\ (Event|Path)*\ Path\ \textbf{\textit{end}} \mid \epsilon$$
$$Path \rightarrow \textbf{\textit{Pid}}$$

Fig. 4. Intraprocedural control flow trace grammar. ***Pid*** is identifier for paths.

existence of loop paths leads to the control flow of an execution unit being represented by a sequence of identifiers instead of just one identifier.

3.3.2. Intrapocedural Control Flow Trace Grammar. Now we extend our high-level execution trace grammar to include the control flow information using the BL encoding. The control flow path taken is recorded in between the execution unit identifier and the corresponding ***end*** symbol in the trace. We call this an intraprocedural control flow trace. Note that the execution units that have loops would record a sequence of identifiers representing the loop paths taken during each iteration. For example, let event *E1* be the procedure in Figure 3(b), which has a loop, and assume the loop iterated three times. The following intraprocedural trace represents such an execution. The first loop iteration took loop path **1**, and the next two iterations took loop path **3**. These loop paths are recorded at the end of the loop. Path **2** from the loop exit to the end of the procedure is taken at the loop exit and is recorded at the end of the procedure along with the ***end*** symbol.

$$\overbrace{\textit{E1}\ \mathbf{1\ 3\ 3\ 2\ end}}.$$

To illustrate that the trace not only records the detailed paths taken but also narrows down the places where the preemption occurs, we consider the following example. Let event *E2* be the simple procedure in Figure 3(a). As in the proceeding example, let event *E1* be the procedure in Figure 3(b) and assume the loop executed three times. The following intraprocedural control flow trace shows an interleaved execution of events $E1$ and $E2$. From the trace, it is clear that event *E2* interrupted event $E1$'s execution during the third iteration of the loop.

$$\overbrace{\textit{E1}\ \mathbf{1\ 3}\ \overbrace{\textit{E2}\ \mathbf{1\ end}}\ \mathbf{3\ 2\ end}}\ .$$

We observe that the intraprocedural control flow trace still retains the nested structure and can be described by a context-free grammar. The grammar is presented in Figure 4. 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.

3.4. Interprocedural Control Flow Tracing

The primary execution units (tasks or 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*. In this section, we first present the challenges of interprocedural control flow tracing and then present a novel static program analysis based on interprocedural summary analysis to compute interprocedural control flow paths.

$$Trace \rightarrow EUnit* \qquad EUnit \rightarrow Id\ (Func|Event|Path)*\ Path\ \textbf{\textit{end}} \mid \epsilon$$
$$Id \rightarrow \textbf{\textit{Tid}} \mid \textbf{\textit{Eid}} \qquad Func \rightarrow \textbf{\textit{Fid}}\ (Func|Event|Path)*\ Path\ \textbf{\textit{end}} \mid \epsilon$$
$$Path \rightarrow \textbf{\textit{Pid}} \qquad Event \rightarrow \textbf{\textit{Eid}}\ (Event|Path)*\ Path\ \textbf{\textit{end}} \mid \epsilon$$

Fig. 5. Interprocedural control flow trace grammar. ***Fid*** is identifier for functions. A new nonterminal *Func* is introduced to represent function calls.

3.4.1. Challenges. Since function calls exhibit the same nesting property as execution units, that is, 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 5, 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 grammar in Figure 5. 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 following trace.

$$\overbrace{\overbrace{T1\ \textbf{2 end}}\ E1\ F\ \overbrace{G\ \textbf{1 end}}\ \overbrace{E2\ \textbf{1 end}}\ \textbf{1 end}}^{}\ \textbf{0 end}.$$

The problem with the simple approach is that TinyOS applications often have a large number of small functions. According to the grammar in Figure 5, a function identifier, a special ***end*** symbol, and at least one path identifier 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.

3.4.2. Approach. Our approach focuses on representing the interprocedural control flow path of the entire execution unit with a single identifier instead of a sequence of identifiers for individual function calls. This approach can reduce the number of bytes recorded in the trace significantly, as only one identifier 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 interprocedural path encoding, which depends on the invocation points. For instance, in Figure 6, there are two invocation sites of function *Bar*—*B* and *G*. In order to produce correct interprocedural encoding regarding call site *G*, the path identifier counter should be increased by 1 along the edge $J \rightarrow M$, because there is only one interprocedural path leading from *M*'s left sibling *K* to the exit node of function *Foo*. However, when the call site *B* is considered, the path identifier counter has to be increased by 4 along the edge $J \rightarrow M$, because there are four interprocedural paths from *K* to the exit node of function *foo* (partially induced by the predicate inside function *Bar* called at *G*). 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 runtime, 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 that the value p is returned by the function at runtime. 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 runtime path identifier 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 $n \times x$ (interprocedural) paths leading from the call

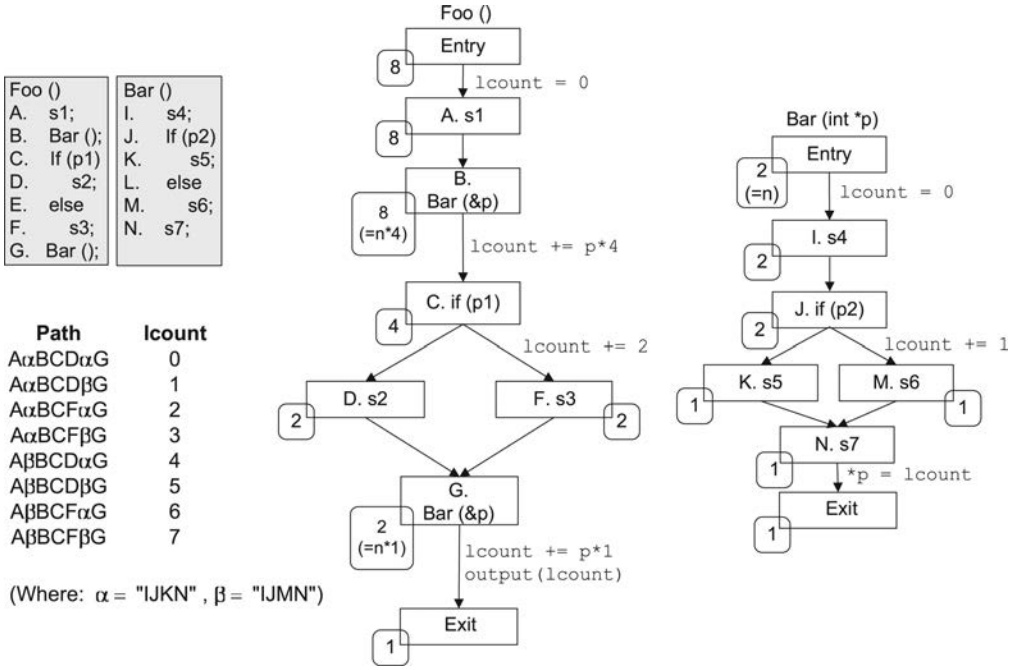


Fig. 6. Example for the modular interprocedural paths computation algorithm.

site to the end; note that 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 interprocedural paths led by the callee path p and trailed by one of the x paths in the caller.

The interprocedural instrumentation algorithm is presented in Algorithm 3. The algorithm instruments a given function f, taking care of the encoding of the functions called by function f. We assume that the functions called inside function f, referred to as function g in Algorithm 3, do not have loops or recursion. Otherwise, the individual function encoding is employed. Note, however, that function f itself can have loops or recursion. We explain the benefits of using individual function encoding when the callees contain loops or recursion in Section 3.4.4.

Similar to Algorithm 2, the annotation of a node represents the number of (interprocedural) 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 function g, is traced (i.e., needs to be instrumented) and does not have loops or recursion, the paths inside g will be encoded as part of the path identifier of f in a fashion as previously explained. The call to `retrievePath` in the instrumentation is provided as part of the tracing library to get the aforementioned p, that is, the runtime path inside g. More particularly, the call retrieves the variable `lcount` of g, which is a local variable on function g's stack frame, encoding the path just taken inside function g. Note that the value is not destroyed as the execution just returns from function g. The `retrievePath` can also

ALGORITHM 3: Instrumenting a Function f.
Assume all function invocations have been made independent statements instead of subexpressions, as in a low-level intermediate representation, that is, each call site has only one successor; all the loop back-edges in function f have been replaced with dummy edges, as described earlier. Function `retrievePath` retrieves the path identifier of the callee at runtime.

```
procedure InterProcPath (f)
  annotate exit node with 1
  for each node s in function 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 function g is traced and does not have loops or recursion then
        n ← the number of static paths of function g
        s.annotation ← n × x
        instrument edge s → s.succ[0] with "lcount+=retrievePath (g)×x"
      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 "lcount+=sum"
      end for
    end if
  end for
```

be implemented as a parameter to the callee, which returns the path taken in that parameter, such as a pointer variable, as shown in Figure 6.

3.4.3. Illustration. We use the example in Figure 6 to illustrate the idea. Each node is annotated with the number of interprocedural paths from that node to *Exit*. The annotations are shown as boxes with rounded corners. The edges are annotated with path identifier increments.

The function *Bar* has two paths, namely α and β, and the value of n is 2. The number of paths to exit from G's successor is 1 ($x = 1$), and therefore the number of paths from G to exit is 2 ($n \times x = 1 \times 2$), which is shown as G's annotation. Now this value is propagated upwards in the control flow graph of function *Foo*. The number of paths to exit from B's successor, C, is four ($x = 4$), which includes the two possible paths inside function *Bar* called at G. The annotation on call-site node B is therefore eight ($= 4 \times 2$). The edge increments for edges $B \rightarrow C$ and $G \rightarrow Exit$ are annotated with $4 \times p$ and $1 \times p$, where p can be 0 or 1 depending on the path inside function *Bar* taken at runtime. The set of interprocedural paths and their encodings are shown in Figure 6.

3.4.4. Loops and Recursion. We observe that loop paths can be interprocedural if the loop is in a called function or if a function is called inside the loop. To reduce the trace size, it is important to use fewer bits to record loop paths, as they get recorded at each iteration of a loop. Recursions are similar to loops, because at the end of every recursive call, the path taken has to be recorded in the trace. Recursion can be handled in the same way as loops are handled.

Our approach to reducing the number of bits required to record loop paths is to capture the context of the loops in a function once and use identifiers local to that function for recording the loop paths. The context of all the loops in a function is

captured by recording the function identifier and the ***end*** symbol at the start and end of the function, respectively. For example, if function *Bar* had loops in Figure 6, then the control flow paths of *Bar* would be recorded independent of its caller function *Foo*. The control flow paths including loop paths would be encoded with identifiers local to function *Bar*. An example trace is shown here.

$$\overbrace{\textit{Foo}\ \overbrace{\textit{Bar}\ \mathbf{0\ 1\ 1\ 2\ 1\ 3\ end}}\ \mathbf{8\ end}}.$$

By recording the context of the loops, the identifiers for the loop paths are drawn from the local (intraprocedural) namespace, which can be encoded with few bits. If the context of the loops is not captured, the global (interprocedural) namespace has to be used for encoding the loop paths. While global identifiers make every path including loop paths unique across the program, the global namespace may become very large for any nontrivial program. The identifiers drawn from larger global namespaces require more bits to encode than the identifiers drawn from local namespaces. The idea of using local namespaces for logging has been shown to be advantageous in WSN settings [Shea et al. 2010].

Melski and Reps [1999] proposed an interprocedural control flow path profiling algorithm, referred to as MR algorithm, based on the BL algorithm. The MR algorithm builds a super graph, that is, an interprocedural control flow graph, and computes 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 calling context. The function parameters are used to pass context-sensitive coefficients of the linear functions. In other words, different coefficients are used for different contexts. The MR algorithm produces a globally unique path identifier for every interprocedural path, including loop paths, in the program. The use of global identifiers to record loop paths requires more bits per loop path recorded in the trace.

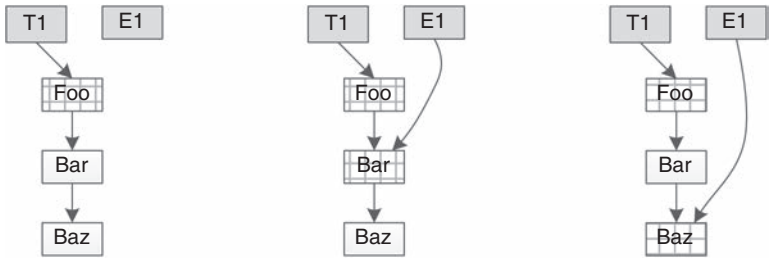
Our approach is compared with the MR algorithm quantitatively in Section 6.5.2 and qualitatively in Section 8. The identifiers used in our approach to record trace entries require fewer bits to encode than the MR algorithm. We show in Section 6.5.2 that the trace size obtained with the MR algorithm is up to 35% larger than the trace size generated by our approach when the identifier in the MR algorithm uses one more byte than our approach for encoding.

In TinyOS applications, loops are infrequent, and we found that the number of iterations in most loops is small and that the control flow paths inside these loops are usually 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 are no branches inside the loop. In TinyOS, recursion is almost nonexistent, perhaps due to potential stack overflow.

3.5. Tracing Event-Driven Programs

Algorithm 3 is generic and can be applied to instrument any C program. It is not straightforward, however, to apply the algorithm when parts of an event-driven concurrent program, such as a TinyOS application, is traced. When only a subset of functions in a program is traced, it is important to handle cases, such as untraced callers and callees of a function, so that interprocedural paths of all and only traced functions are recorded. Event-driven concurrency add further complexity as a function can be called from multiple execution units.

In this section, we present an algorithm for tracing a subset of functions in an event-driven concurrent program, such as a TinyOS application. We first describe the two



(a) Case 1. (b) Case 2. (c) Case 3.

Fig. 7. Example call graphs showing different call edges yield different sets of independent and dependent functions. The set of traced functions (S_T) are *Foo*, *Bar*, and *Baz*. The independent traced functions (S_I) are shown as boxes filled with patterns, and the dependent traced functions (S_D) are shown as boxes not filled with patterns.

types of traced functions and an algorithm for classifing the set of traced functions into these two types. We then present an algorithm that orders the set of traced functions and instruments them correctly using Algorithm 3.

3.5.1. Types of Traced Functions. The set of traced functions, S_T, can be divided into two mutually exclusive sets, namely, *independent* traced functions, S_I, and *dependent* traced functions, S_D.

Independent traced functions are those whose control flow path cannot be subsumed into the caller's control flow path. Traced functions containing loops or recursion as well as functions with untraced callers are independent. For independent traced functions, the identifier of the function and the interprocedural control flow paths taken inside that function are recorded in the trace. Dependent traced functions are those whose control flow path is subsumed by the caller's interprocedural control flow path. These functions are not included in the trace explicitly. We observe that the following are true about S_I and S_D; $S_T = S_I \cup S_D$ and $S_I \cap S_D \leftarrow \emptyset$.

Algorithm 3 expects that all the functions called in a given function *Foo* are instrumented before instrumenting function *Foo* itself. Since the callees in function *Foo* in turn might have other callees, the transitive closure of functions of all the callees inside *Foo* have to be instrumented before instrumenting *Foo* itself. Therefore, the set of traced functions has to be ordered before applying Algorithm 3.

In an event-driven concurrent system like TinyOS, naturally, the main function and the execution units (i.e., tasks and events) are independent as they start new executions. Since TinyOS is concurrent, a function can be called from different execution units. Therefore, to identify the callers of a function in a program, the static call graph has to be traversed from each execution unit.

Figure 7 illustrates the different sets of callers arising from potentially concurrent events. Task *T1* and event *E1* show the start of different executions in the call graph, and these are not traced. The set of traced functions (S_T) consists of functions *Foo*, *Bar*, and *Baz*, which do not have loops or recursions. The three cases show different call dependences for the traced functions and how these dependences change the set of independent and dependent traced functions. The boxes filled with patterns represent the set of independent traced functions (S_I), while the boxes not filled with patterns represent dependent traced functions (S_D). Figure 7(a) describes a simple scenario in which the traced functions are called from only one execution unit. Function *Foo* is independent, as its caller is not traced, while functions *Bar* and *Baz* are dependent, as their callers are traced. In Figure 7(b), in addition to function *Foo*, function *Bar* is independent and has an untraced caller from a different execution unit. Similarly

in Figure 7(c), functions *Foo* and *Baz* are independent. Function *Bar* in Figure 7(c), however, is dependent, because its caller is being traced.

Algorithm 4 classifies a given set of traced functions, S_T into S_I and S_D. The function FINDCALLERS () identifies a set of callers for a given traced function, which traverses the call graph from each execution unit in the program.

ALGORITHM 4: Classifies the Set of Traced Functions S_T into S_I and S_D.
It uses the function FINDCALLERS () that returns all the callers of a function f in an event-driven program. FINDCALLERS () uses a helper procedure FINDPARENT () which takes a call graph, an execution unit, and a function and returns the parent of the function in the path from the execution unit in the call graph. Set S_E is the set of all execution units in the event-driven program.

```
function ClassifyTracedFunctions (S_T)
  for each function f in S_T do
    S_C ← FindCallers (f)
    if function f has loops or recursion then
      S_I = S_I ∪ f
    else if S_C ⊆ S_T then
      S_D ← S_D ∪ f
    else
      S_I ← S_I ∪ f
    end if
  end for
  return ⟨S_I, S_D⟩

function FindCallers (f)
  build call graph cg
  S_C ← ∅
  for each function e in S_E do
    S_C ← S_C ∪ { FindParent (cg, e, f) }
  end for
  return S_C
```

3.5.2. Instrumenting Event-Driven Programs. We observe that a function *Foo* can be instrumented using procedure INTERPROCPATH () in Algorithm 3 only after all its callees in S_D are instrumented. For callees in S_I, the ordering is not important because their control flow path is recorded independent of the callers. Therefore, functions in S_D have to be instrumented before functions in S_I. Furthermore, a function in S_D can be instrumented only after its transitive closure of functions of all the callees in S_D are instrumented. This can be achieved by considering the functions in S_D in the reverse topological order of the call graph of functions in S_D. Once the procedure INTERPROCPATH () is applied to all functions in S_D, it can be applied to all functions in S_I in any order.

After applying procedure INTERPROCPATH (), the dependent and independent traced functions are instrumented differently. The dependent traced functions are instrumented to return the control flow path taken at runtime to the caller. This is implemented by adding a new output argument, that is, a pointer variable in C (`p`) to dependent traced functions, which gets assigned to the control flow path at the end of the function at runtime, whereas the independent traced functions are instrumented to record the function name and the interprocedural control flow path taken at runtime into the trace, and in addition, the loop exit nodes are instrumented to record the loop paths into the trace. The steps described are shown formally in Algorithm 5.

ALGORITHM 5: Instruments the Set of Traced Functions S_T in an Event-Driven Concurrent Program. `id` refers to the function identifier.

```
⟨ S_I, S_D ⟩ ← ClassifyTracedFunctions (S_T)
for each function f in the call graph of S_D in the reverse topological order do
    InterProcPath (f)
    instrument the exit node with "*p = lcount"
end for
for each function f ∈ S_I do
    InterProcPath (f)
    instrument the entry node with "trace.print(id)"
    instrument the exit node with "trace.print(lcount)"
    for i = 1 to the number of loops in f do
      instrument the loop exit node with "trace.print(lcount); lcount = 0"
    end for
end for
```

4. IMPLEMENTATION

We implemented our algorithms for both TinyOS 1.x and TinyOS 2.x in a tool called *TinyTracer* [Sundaram et al. 2011, 2010]. We tested TinyTracer on mica2 motes. We used the CIL source-to-source transformation tool for C to instrument the C code generated by the nesC compiler version 1.3. The instrumented code is then compiled to create an executable that can run on motes as well as on emulators, such as Avrora [Titzer et al. 2005], ATEMU [Polley et al. 2004], or simulators such as TOSSIM [Levis et al. 2003]. The trace is recorded in the flash at runtime and can be retrieved for later use. We also developed a trace parser using Python to pretty-print the compressed trace. Our implementation is available at `http://sss.cs.purdue.edu/projects/TinyTracer`.

TinyTracer has two core components: (1) the TinyTracer engine, a compile-time CIL module that does interprocedural analysis and automatically instruments the code, and (2) `TinyTracerC`, a runtime nesC component that records the trace, compresses it, and either commits it to the flash or sends it to the base station.

4.1. TinyTracer Engine

The TinyTracer engine is at the heart of the TinyTracer implementation. Given a set of functions or components to trace, the TinyTracer engine performs interprocedural summary analysis, as described in Section 3, and automatically instruments the code to trace these functions at runtime.

We note that the nesC structures (i.e., components, tasks, events) are lost in the translation from nesC to C. However, it is possible to create those structures by analyzing the C file itself. Execution units and component names have to be identified from the C file to apply Algorithm 5. The nesC components can be identified readily because the component names are prefixed to the function names in those components. Tasks can be identified by examining the argument to the `TOS_Post(foo)` function when each task is posted. Interrupts can be identified since interrupt handler names begin with `__vector`. Because of the nesC rule that only asynchronous functions can call other asynchronous functions, the *transitive* closure of functions called by interrupt handlers comprise all asynchronous functions in a nesC program.

4.2. `TinyTracerC`

`TinyTracerC` is the runtime nesC component that is responsible for three phases of tracing: trace recording, trace compression, and trace storage or transfer. This component handles the tricky issues, such as data races and buffer overflows that arise in

different phases, by carefully managing limited resources. We elaborate on the these three phases next.

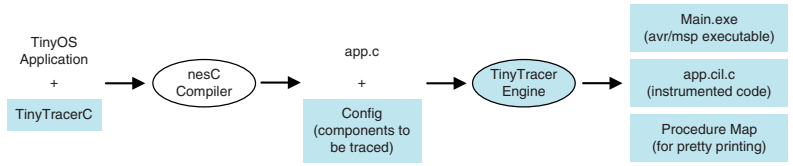
4.2.1. Trace Recording. The trace generated is recorded at runtime in two in-memory trace buffers, each of length 192 bytes. The buffer sizes depend on the application being traced and are, therefore, configurable. However, for most programs, a trace buffer of size 96 bytes and eight flash pages each of length 16 bytes are enough to capture the trace generated by the application. Since the trace is constantly generated at different speeds, it is crucial to handle trace-buffer corruption. This situation is similar to the classic producer-consumer problem, where the trace compression and trace storage phases are the consumers and the trace generated by the execution is the producer. `TinyTracerC` carefully coordinates the buffer usage with locks. Furthermore, since multiple threads could record trace at the same time, trace recording is made *atomic*. During runtime, when one of the trace buffers is filled, the trace generated is recorded in another buffer, and a TinyOS task, `CompressTrace`, is posted on the full buffer to compress the filled buffer.

4.2.2. Trace Compression. The `CompressTrace` task runs in the background and compresses the buffer using an effective multistep, custom-tailored compression technique, which is based on the following key observations. TinyOS application executions are repetitive, and a path is often exercised many times, giving rise to a large number of repeated subsequences in the trace. Note that such a repetition is hardly exploited by the encoding scheme we presented in Section 3, which relies on program structure instead of runtime 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 identifiers, even though these identifiers may be the same.

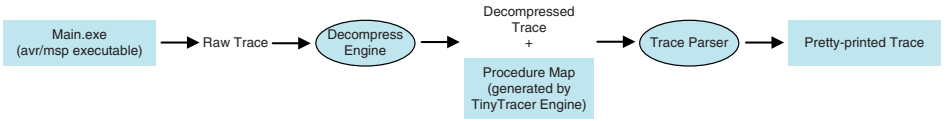
For normal scenarios where the resources are not so constrained, this repetition can be easily captured by the LZW compression algorithm [Welch 1984] or Sequitur [Larus 1999]. However, these are not applicable for resource-constrained WSNs. Fortunately, the repetitive patterns are simpler in our case because of the repeating sequences of events as well as the small number 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. The occurrences of this subsequence are replaced with a special symbol in the trace, and the second-most frequent subsequence in the compressed trace is identified. Note that the second-most frequent subsequence might contain the special symbol of the most frequent subsequence. The mined patterns are used by the `CompressTrace` task at runtime for compression. More particularly, the `CompressTrace` 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 performs run-length encoding, which replaces the repeating characters with the length of the repetition and the character itself.

We opted for this approach as opposed to mining patterns at runtime to reduce the runtime overhead and the related memory overhead. As shown in Section 6.4, this compression is quite effective, while incurring much less overhead compared to LZW [Welch 1984] or other similar compression techniques that mine patterns on the go. Furthermore, our multistep compression approach is amenable to state-machine implementation that relinquishes CPU between steps. This is an important consideration in the design as TinyOS expects tasks to be as short as possible [Levis et al. 2005]. Long-running tasks can possibly starve other executions including trace storage. Therefore, the `CompressTrace` task is implemented as a state machine that relinquishes CPU and posts itself after each step.



(a) Compiletime.



(b) Runtime.

Fig. 8. Workflow of TinyTracer at compile time a and runtime b for TinyOS applications. The shaded components are parts of or outputs produced by TinyTracer.

4.2.3. Trace Storage and Transfer. The compressed trace is stored into flash pages by the `CompressTrace` task. The flash pages, each of length 16 bytes, are drained by the `EEPROM` component in TinyOS 1.x and are written to flash memory. `TinyTracerC` also provides an option of sending the trace over the radio or the serial port by simply changing a compiler flag. When transporting the trace to the base station, techniques such as a separate radio stack [Tolle and Culler 2005] or assigning message priorities [Sundaram et al. 2008] can be used to transport it in a reliable, energy-efficient way.

4.3. Usage

The compile-time workflow of TinyTracer is shown in Figure 8(a). To trace a TinyOS application, the developer has to create a configuration file containing the names of the nesC components or functions. As a first step, the developer links the runtime component `TinyTracerC` with the TinyOS application by wiring the `StdControl` interface of the application's main component to the TinyTracer runtime component. The nesC compiler produces a C file (`app.c`). The developer creates a configuration file containing the components and functions to be traced. This configuration file along with `app.c` is given as input to the TinyTracer engine. The TinyTracer engine produces a new C file (`app.cil.c`) that contains the instrumented code and a procedure map file, which is used for pretty-printing the trace later. The TinyTracer engine can be configured to use the target platform compiler to compile `app.cil.c` and get the target platform specific executable.

The runtime workflow of TinyTracer is shown in Figure 8(a). The raw trace collected is decompressed and parsed by the trace parser, which uses the procedure map file to pretty-print the trace.

An example of a pretty-printed trace for the `Oscilloscope` application in TinyOS is shown in Figure 9. The raw trace collected is shown as follows as two-byte hexadecimal numbers. The compressed part is $0x0C80$, and it is decompressed as $0x0030$ $0x0005$ $0x0056$ $0x0029$. Since there are no loops or event preemptions in this example, the decompressed trace should be read from left to right as pairs, each pair consisting of procedure identifier and the path taken inside that procedure.

$$0x0C80\ 0x0C80\ 0x0C80\ 0x0C80\ 0x0C80\ 0x0040\ 0x0005\ 0x0020\ 0x0005.$$

```
 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 repeat 9 times]
 6 <OscilloscopeM__dataTask> start
 7   <LedsC__Leds__yellowToggle> start
 8   <LedsC__Leds__yellowToggle> end 0
 9 <OscilloscopeM__dataTask> end 0
10 <OscilloscopeM__DataMsg__sendDone> start
11 <OscilloscopeM__DataMsg__sendDone> end 0
```

Fig. 9. Partial listing of pretty-printed trace.

5. CASE STUDIES OF COMMON FAULTS

In this section, we identify four common faults that occur in TinyOS, namely, initialization faults, split-phase faults, state machine faults, and task queue overrun faults. These faults have been reported several times in the literature [Yang et al. 2007; Keller et al. 2009] as well as in mailing lists. They also played a crucial role in the design of TinyOS 2.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 of TinyOS applications. We uncover one previously unknown bug in the widely used flash/EEPROM component in TinyOS 1.x.

In addition to the four common faults, we discuss a network fault that was diagnosed using the traces from TinyTracer. Furthermore, for each case study, we present a user study that reports the time taken for a user to analyze the traces and find the root cause as well as the amount of traces collected. The spurious code statements contributing to the fault are followed by `<-- delete`; missing statements to be inserted to repair the fault are followed by `<-- insert`, wherever applicable.

5.1. Task Queue Overrun Faults

Task queue overrun faults represent a class of notorious faults in TinyOS 1.x, which has caused serious problems, including continuous reboots [Keller et al. 2009] and deadlocking of the radio stack [Yang et al. 2007]. Task queue overruns can happen if tasks are starved from execution due to long-running tasks [Keller et al. 2009] or high rate of interrupts [Yang et al. 2007].

5.1.1. Repeated Resets in PermaSense. As introduced in Section 1.2, the PermaSense project [Hasler et al. 2008] strives to collect geophysical data with WSNs in the harsh environment of the Swiss Alps. After six months of deployment (March 2009), the deployment had severe performance degradation [Keller et al. 2009]. A soft reset, which reinitialized memory and restarted the application, was placed as a safety mechanism to survive task queue overruns in the field. However, the fault survived soft resets and continued to cause more resets. Indeed, extensive resets of nodes (up to 40 resets per node per day) were observed for three months.

It was quite a challenge to diagnose this fault, as it did not manifest itself in the simulator or lab deployments, and it took six months to manifest in the real deployment. In fact, the fault was localized only after physically recovering some of the affected nodes and analyzing the RAM dumps. The diagnosis required several expensive trips to the mountain top over the span of a few weeks.

Fault Description. The resets were due to task queue overrun, which was caused by a long-running task that blocked the execution of other tasks. The long-running task was a *lookup task*, which mapped a specific block in a file stored in the external SD card (flash) to the physical memory. The external SD card was used to store the sensed

values, and the lookup task was written such that its running time increased with the amount of data stored in the external SD card. This was the reason behind the manifestation of the fault after several months of the deployment, when the external SD card was more than half full. Since the data in the external SD card was preserved across soft resets, the lookup task continued to overrun the task queue even after resets. The fault was fixed by making the lookup task a multistep task that posts itself after every step and using a small cache to avoid the expensive read operation.

Tracing. Tracing the control flow path could have been valuable in this scenario. If the control flow tracing were turned on and the trace were sent to the base station immediately after a soft reset, this trace would have clearly shown that the long-running lookup task was executing multiple expensive I/O operations in a loop. This would have helped the developer determine why that particular task was executing for a longer time than expected and causing a task queue overrun.

The state-of-the-art of debugging techniques, such as Clairvoyant [Yang et al. 2007], could have helped diagnose this fault, but it is not clear how much messaging overhead would have been involved. Function call traces generated by techniques, such as NodeMD [Krunic et al. 2007] and LIS [Shea et al. 2010], could have hinted the problem; however, they would not have clearly explained the I/O interactions (due to lack of control flow information), so much programmer effort would have been required to localize the error.

5.1.2. Deadlock in CC1000 Radio. 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. [2007] showed how Clairvoyant could have helped diagnose this tricky fault.

Fault Description. The code snippet is shown in Figure 10. The `TXSTATE_DONE` case in the SPI interrupt handler represents successful transmission, and the actions that need to be taken are to wait for a short while, intimate the application (i.e., `post PacketSent`), and put the radio in receiving mode (i.e., `SpiByteFifo.rxMode` and `CC1000Control.RxMode`). As shown by lines 14–17 in Figure 10, the functions `SpiByteFifo.rxMode` and `CC1000Control.RxMode` are called before posting the `PacketSent` task. Since these two function calls invoke `wait`, the execution path takes a relatively long time (around 400 microseconds). The SPI interrupt rate was so high that it prevented the task from executing. The repeated posting and starvation led to task queue overflow and, consequently, to failure of posting the `packetSent` task, which meant the radio state was not changed to `IDLE_STATE` and the radio deadlocked. To fix the fault, the radio should be put in receiving mode only after intimating the application.

Tracing. The control flow trace of the CC1000 radio component execution would show the repeated invocations of the SPI interrupt handler with the path traversing along the case `TXSTATE_DONE` and encountering a failure in posting the `PacketSent` task. This is a clear indication of a task queue overrun. Without the trace, a substantial amount of manual labor is required to detect the fault. Function call traces generated by NodeMD [Krunic et al. 2007] and LIS [Shea et al. 2010] could have helped as well.

5.2. 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, as we discuss next through one such case study.

```
1  task void shortDelay() {
2    TOSH_uwait(10);
3  }
4  void postDelay() {
5    for(i=0;i<8 i++)
6      post shortDelay();
7  }
8  async event result_t SpiByteFifo.dataReady(uint8_t data_in) {
9    ...
10   case TXSTATE_DONE:
11   default:
12     if (bTxPending == TRUE)
13       postDelay();
14     call SpiByteFifo.rxMode();       <-- delete
15     call CC1000Control.RxMode();     <-- delete
16     bTxPending = FALSE;
17     if (post PacketSent()) {
18       call SpiByteFifo.rxMode();     <-- insert
19       call CC1000Control.RxMode();   <-- insert
20       RadioState = IDLE_STATE;
21     }
22     ...
23 }
24 event result_t Timer.fired() {
25   post processing();
26   return SUCCESS;
27 }
28 task void processing() {
29   call Send.send(TOS_BCAST_ADDR, sizeof(IntMsg), &readyMsg);
30 }
```

Fig. 10. Partial listing of CC1000 radio stack code.

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

Fault Description. TOSSIM dumped the core after running one of the benchmark programs for a while and exiting. Since TOSSIM is a discrete event simulator, the `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. The problem turned out to be a subtle fault in the EEPROM component, as described later.

Tracing. We traced all the components used, including radio and flash, and the trace was printed 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` had inserted a simulator event, which had a null handler. The handler was assigned when the EEPROM was initialized. Since we forgot the wiring of the 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.

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

Fig. 11. Partial listing of Flash/EEPROM component in `tos/platform/mica/eepromM.nc`.

The diagnosis would have been simple had the EEPROM failed at `startWrite`, as the debug output would have pointed to the EEPROM component directly. The diagnosis became tricky because the EEPROM had a fault that had not been discovered, 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 11. The programmer who had checked for an idle state before starting a write, however, did not anticipate that the system could be in an idle state bypassing the regular initialization scheme. The problem was due to the `enum` used to maintain the state, which by default, was assigned a value of zero. To fix the bug, an additional state *uninitialized* has to be created, as shown in Figure 11, which prevents any uninitialized access to the component.

Most of the state-of-the-art debugging techniques are not applicable to this bug except for tracing technique, such as NodeMD [Krunic et al. 2007] and LIS [Shea et al. 2010], which could have helped with much programmer effort.

5.3. Split-Phase Operation Faults

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

5.3.1. Poor Throughput in LEACH. We describe a subtle fault due to a high-level data race which occurred in an implementation of the LEACH protocol [Heinzelman et al. 2002].

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, the 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, upon receiving the advertisement messages from

```
1  task sendDebugMsgTask() {
2    atomic tempFlag = radioSendBusy;
3    msgP->type = debug_type;    <-- delete
4    if (tempFlag)
5      return;
6    else
7      atomic radioSendBusy = true;
8    ...
9    msgP->type = debug_type;    <-- insert
10   if (call Send.send(addr, length, msgP) !=SUCCESS)
11     atomic radioSendBusy = false;
12 }
13 task sendTDMAScheduleTask() {
14   // calls send message, details omitted
15   ...
16   if (call Send.send(addr, length, msgP) != SUCCESS)
17     atomic radioSendBusy = false;
18 }
19 event result_t StateTransitionTimer.fired() {
20   // changes state when timer fires
21   switch (curState) {
22     case SEND_TDMA_SCHEDULE:
23       post sendTDMAScheduletask();
24       break;
25     case INFORM_BASE_STATION:
26       post sendDebugMsgTask();
27       break;
28       ...
29   }
30   ...
31 }
32 event result_t SendTDMASchedule.sendDone(TOS_MsgPtr msg, result_t success) {
33   atomic radioSendBusy = false;
34   ...
35 }
```

Fig. 12. Partial listing of LEACH protocol implementation.

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.

Fault Description. The implementation of LEACH used a finite state machine in which the state that represented the stage that the protocol was executing in, for example, SEND_TDMA_SCHEDULE. A partial listing of LEACH code is given in Figure 12. The implementation was run on TOSSIM using a CC1000 bit-level radio. When the number of nodes was small (i.e., 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, since the debug logs at the cluster heads showed successful sending of TDMA scheduling messages, the reason for packet drop at the receiver was unclear.

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 in progress. The control flow in the trace disclosed that before checking the radio sending flag, the message type was modified by the cluster head. 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 [Cooprider et al. 2007]. Thus, tracing the control flow can be very helpful in tracking high-level data race conditions, such as the one just explained.

5.3.2. Unusable Resources in `DeferredPowerManagerP`. When split-phase operations are used, it is important to make sure failure return values are handled correctly. If the start of a split-phase operation does not succeed, the state of the system should be restored to the state before the start. Failure to do so could block the system, as the callback event of the split-phase operation would never happen to restore the system's state. This is a fairly common problem with split-phase operations. We discuss a representative bug found in the TinyOS bugs database. This fault was reported two years after its release and fixed recently, according to the TinyOS CVS. We present a generalization of this representative fault after discussing the fault.

Deferred Power Manager. The `DeferredPowerManagerP` is a component in TinyOS 2.x that implements a power management policy using a *deferred* power control scheme, whereby devices are powered-on immediately after being requested, but powered-off after some small delay from being released. The component uses the `SplitControl` interface to start and stop the device managed. The partial code listing of the component along with the `SplitControl` interface is shown in Figure 13. The user of the `SplitControl` interface must call command `start` to power a device on and command `stop` to power a device off. The calls to either command return one of `SUCCESS`, `FAIL`, `EBUSY`, and `EALREADY`. When the `start` command is called and the device is off, it moves into the starting state and returns `SUCCESS`. Later, when it is fully started, it would call back with the event `startDone`. If the device is already on when the `start` command is issued, it returns `EALREADY`. Similarly, these semantics also apply to the `stop` command. More details of the power management policy and the split control interface can be found in TEP115 (http://www.tinyos.net/tinyos-2.x/doc/html/tep115.html).

Fault Description. The `DeferredPowerManagerP` is the default owner of the devices it manages. When a user requests a device from `DeferredPowerManagerP`, it calls `SplitControl.start` to power the device on. It releases the device when a `SplitControl.startDone` event occurs so that the user can take control of the device. If the device was already on, `SplitControl.start` returns `EALREADY`, as

```
1  interface SplitControl {
2    command error_t start();
3    event void startDone(error_t error);
4    command error_t stop();
5    event void stopDone(error_t error);
6  }
7  task void startTask() {
8    call SplitControl.start();                    <-- delete
9    if (call SplitControl.start() == EALREADY)    <-- insert
10     call ResourceDefaultOwner.release();        <-- insert
11   ...
12 }
13 event void TimerMilli.fired() {
14   call SplitControl.stop();                     <-- delete
15   if (call SplitControl.stop() == EALREADY)     <-- insert
16     signal SplitControl.stopDone(SUCCESS);      <-- insert
17   ...
18 }
```

Fig. 13. Listing of `SplitControl` interface and partial listing of `DeferredPowerManagerP` component.

it is supposed to, but the `DeferredPowerManagerP` does not release the device. Note that in this case, there would not be a `SplitControl.startDone` event callback, and hence, the device would not be released at all. To fix this fault, the `DeferredPowerManagerP` component should check for the return value of the `SplitControl.start` and release the device immediately if the return value is `EALREADY`.

The fault has not been reported for two years after the faulty version was released. The reason being that the case where the device is already on is rare, and when it does happen, it is tricky to diagnose just by looking at the `DefaultPowerManagerP` code. A cursory look at the code suggests that the `DefaultPowerManagerP` component in fact releases the device in `startDone` as expected.

Tracing. The control flow trace for `DefaultPowerManagerP` would show a call to `SplitControl.start` and the control flow paths taken inside. The trace would disclose that `EALREADY` was the return value, meaning that the device was already on and `startDone` would never be called in this case. This would clearly pinpoint that component `DefaultPowerManagerP` does not release the device correctly.

The fault just discussed is only a representative one. We can generalize this fault. TinyOS applications have multiple layers of abstractions, such as the device layer, power management layer, and application layer, and if a service is implemented as split-phase in a particular layer, every layer above it has to implement it as split-phase. For example, sending a message uses a split-control interface to send a packet. The packet is passed through the network layer which uses split-phase operations, then to the MAC layer in which each byte uses split-phase operations, and finally to the hardware which uses a split-phase operation for every bit. If a TinyOS component using a split-phase operation gets blocked at a particular layer, its effect is visible on the layers above. Therefore, it is important to locate the fault in the layer which is responsible and the condition that triggered the fault. If control flow tracing is turned on, it would tell the sequence of commands and events executed as well as the paths taken, which would quickly pinpoint the location of these faults. Therefore, it is clear that control flow tracing is very helpful for these faults.

5.4. State Machine Faults

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 fault. There have been several examples

```
1  async command result_t Voltage.getData() {
2    uint8_t tmpState;
3    atomic tmpState = state;
4    if (tmpState != S_IDLE)
5      return FAIL;
6    if (call ADC.getData() == FAIL)
7      return FAIL;
8    atomic state = S_GET_DATA;
9    return SUCCESS;
10 }
11 void signalReady(uint16_t v) {
12   signal Voltage.dataReady(v);
13 }
14 task void signalReadyTask() {
15   uint32_t tmpVoltage;
16   atomic {
17     tmpVoltage = (uint32_t)1223*1024;
18     tmpVoltage /= voltage;
19   }
20   atomic state = S_IDLE;   <-- insert
21   signalReady (tmpVoltage);
22 }
```

Fig. 14. Partial listing of VoltageM component.

of faulty state transitions discussed in the literature [Kothari et al. 2008] and TinyOS mailing lists. Several thread abstractions [Dunkels et al. 2006; Klues et al. 2009] on top of event-driven programming models have been proposed to alleviate the problem. Note that the fault we discussed in Section 5.2.1 is also an instance of this type.

5.4.1. Unusable Voltage Sensor in `VoltageM`. Here we describe a simple component in which a faulty state transition renders the component unusable after the first use. This fault has been reported in the TinyOS bugs mailing list (http://sourceforge.net/tracker2/index.php?func=detail&ai d=1221595&group_id=28656&atid=393934) and is in the CVS snapshot of TinyOS version 1.1.13. It was reported a few months after the release and was fixed in the later versions.

Fault Description. The `VoltageM` component in TinyOS 1.x measures the supply voltage in mica2/micaz motes. A code snippet of this component is shown in Figure 14. The component has two states, namely, `S_IDLE` and `S_GET_DATA`. Once the `VoltageM` component is initialized, any calls to the command `getData` would result in a callback event `dataReady` containing the voltage value. When the command `getData` is called, the `VoltageM` component issues a command to the lower layer to obtain voltage, but only if it is in `S_IDLE` state. After issuing the command, it sets its state to `S_GET_DATA`. When it gets a callback from the lower layer, it signals the event `dataReady` to the upper layer that called the command `getData`, by posting a task. However, the faulty implementation of this component does not reset the state to `S_IDLE` after signaling to the upper layer. Thus, further calls to the command `getData` return a failure to the user of the component. The fault can be easily fixed by setting the state to `S_IDLE` in the task signaling the event `dataReady`.

Tracing. With control flow tracing, the trace would show the expected control flow for the first request to get data from the `VoltageM` component in which the `getData` calls the lower layer to obtain voltage data. However, further requests would follow the control flow in which `getData` returns a failure because the program is not in the state `S_IDLE`. Analyzing the first few requests' control flow, the programmer could deduce that the task that signals event `dataReady` does not return the component to `S_IDLE` state.

In this particular case, the fault is easy to localize, perhaps even without a trace, because the state machine is simple and the fault is easy to reproduce. We chose this case study because it was reported in the TinyOS mailing lists. However, it is possible to envisage a complex state machine in which a node executes a faulty transition if some predicate is true. Such state transitions would become obvious when navigating or replaying the control flow trace. We observed such faults when implementing LEACH and several other programs. Therefore, control flow trace could be useful in detecting and isolating invalid state transistions.

We note that the state-of-the-art techniques, such as deriving finite state machines [Kothari et al. 2008], can detect most of these faults at compile time itself. However, the faults (such as in Section 5.2.1) could not have been detected with Kothari et al. [2008], as the state itself was missing.

5.5. Distributed Faults

While TinyTracer is aimed at diagnosing individual node faults, the control flow trace containing message sends and receives collected from multiple nodes could be useful for diagnosing distributed faults. TinyTracer helped us in diagnosing two network faults in our implementation of LEACH [Heinzelman et al. 2002]. We discussed one such fault in Section 5.3.1 in which the a fault at the sender node manifested at the receiver node. We discuss another fault in LEACH that led to network congestion in this section. We, however, note that complex distributed faults that depend on the message ordering require recording additional information, such as logical clocks used to order messages.

5.5.1. Congestion in LEACH. LEACH is a TDMA-based clustering protocol, and it's operations are presented in Section 5.3.1. In this section, we describe a fault that caused congestion in LEACH and, as a result, poor throughput at the base station.

Fault Description. When we increased the number of nodes in our simulation of LEACH to 100, we found that the data rate received at the base station reduced significantly. LEACH is a distributed algorithm which makes nodes randomly elect themselves to be a cluster head in each round. Due to moderate performance of random number generators in TinyOS 1.x, the number of cluster heads can vary widely from the expected number of cluster heads across rounds. When the number of cluster heads are much fewer than the expected number of cluster heads, many nodes attempt to join each cluster head. Due to the small size of the time slot, join messages were colliding. Consequently, fewer nodes successfully joined clusters. The nodes that did not join the cluster in a round remained in `NO_TDMA_STATE`, resulting in lower throughput. To repair the fault, we increased the size of the slot as well as the number of time slots for join messages. Furthermore, we introduced a random exponential back-off mechanism to overcome join message collisions. The issue of allowing enough time (or additional slots) for join messages to reach their cluster heads is not addressed by the designers of LEACH protocol [Heinzelman et al. 2002].

Tracing. When the throughput dropped at the base station, the control flow traces from several nodes were examined. The abnormal control flow in the trace revealed that many nodes were in `NO_TDMA_STATE` after sending join messages to the cluster head. This raised suspicion on the cluster heads on whether or not they sent the TDMA schedule message. Analyzing the traces from cluster heads revealed that join messages from only few nodes were received successfully and thus formed smaller clusters. Since join message send was recorded and not the receipt, the link between cluster node and the cluster head must have failed, either due to congestion or channel corruption. When we made the channels perfect in our simulations, we still observed the same

Table I. Time Taken by a User to Find the Root Cause from Trace Analysis

Bug Name (Location)	Analysis Time (Minutes)	Trace Size per Node (KB)
VoltageM (Section 5.4.1)	5	1
Deferred Power Manager (Section 5.3.2)	15	4
LEACH congestion (Section 5.5.1)	25	8
EEPROM initialization (Section 5.2.1)	30	15
CC1000 radio deadlock (Section 5.1.2)	45	50
LEACH data race (Section 5.3.1)	70	12

result, leading us to conclude that the only possible explanation for link failure is join message collision.

5.6. User Study

We conducted a user study to understand the amount of time it takes to analyze traces and find root causes. The results from a single-user study are shown in Table I. The faults are ordered in increasing order of analysis time. In addition to the analysis time, the table reports on trace sizes per node. Due to the lack of source code, the PermaSense case study discussed in Section 5.1.1 was not included. The results show that the time taken to analyze the root cause varies with the complexity of the fault; for the faults we studied, the analysis time ranges between about five minutes and one hour.

6. PERFORMANCE EVALUATION

We evaluate our approach on various typical applications as well as a large real-world application used as part of the Golden Gate Bridge monitoring project. We observe that our tool incurs low energy and memory overheads and produces viable trace sizes. We present a detailed comparison with the state-of-the-art WSN tracing approaches. In summary, our tool produces smaller traces (up to 87% smaller) while incurring much less energy overheads (up to 79.1% less).

6.1. Evaluation Setup

We used ATEMU [Polley et al. 2004], a cycle-accurate emulator, to measure the energy usage of the CPU, flash, and other components. We obtained traces by writing the flash into a file at the end of the emulation. We verified the results for a few benchmarks from the emulator on our mica2 motes testbed. The metrics, benchmarks, and the state-of-the-art tools used in our evaluation are presented next.

6.1.1. Metrics. The *runtime overhead* consists of (1) CPU cycles required to execute the instrumented instructions and to write to the external flash, and (2) the amount of external storage or trace size. We measure the energy consumed by the CPU and external flash to estimate the runtime overhead. The *compile-time overhead* captures the increase in code size due to instrumentation and increase in RAM due to trace buffers. We measure the program memory and RAM of the executable to obtain the compile-time overhead. In addition to incurring low overheads, tracing approaches should produce small traces to be useful in WSNs. Therefore, we measure trace size and compression ratio.

6.1.2. Benchmarks. We chose five default TinyOS 1.x applications that are widely studied [Cooprider et al. 2007; Shea et al. 2010], as well as a large TinyOS application, LRX, as benchmarks. LRX is a module for reliable transfer of large data, developed as part of the Golden Gate Bridge monitoring project, and is one of the largest nesC components (∼1,300 lines of nesC code) in TinyOS 1.x. We used `SingleHopTest` to drive the LRX module. These six benchmarks are described in Table II, in which LOC refers to the

Table II. The TinyOS 1.x Applications in Our Benchmarks Suite

TinyOS Application (Benchmark)	C LOC	Period (ms)	Description
`Blink` (Blink)	2,061	1000	Toggle the LEDS
`Sense` (Sense)	3,730	500	Samples sensors and displays it on LEDS
`Oscilloscope` (Oscilloscope)	5,956	125	Data collection with high sensing rate
`Surge` (Surge)	11,358	2,000	Data collection with medium sensing rate
`CntToLedsAndRfm` (Count)	8,241	250	Counter that broadcasts and displays count
`LRX` (LRX)	10,015	2,000	Reliable large data transfer application

Note: LOC stands for lines of code.

Table III. A Mapping from the nesC Components Traced to the Number of C Functions in Those Components

Benchmark	Number of nesC components traced				
	1	2	3	4	5
Blink	5	9	19	-	-
Sense	7	14	22	31	-
Oscilloscope	11	18	25	34	42
Surge	11	18	27	37	47
Count	6	20	26	39	48
LRX	29	39	53	63	-

number of lines in the C file generated by the nesC compiler. Although our tool is implemented for both TinyOS 1.x and TinyOS 2.x, we chose TinyOS 1.x and mica2 motes due to the availability of evaluation tools, such as ATEMU.

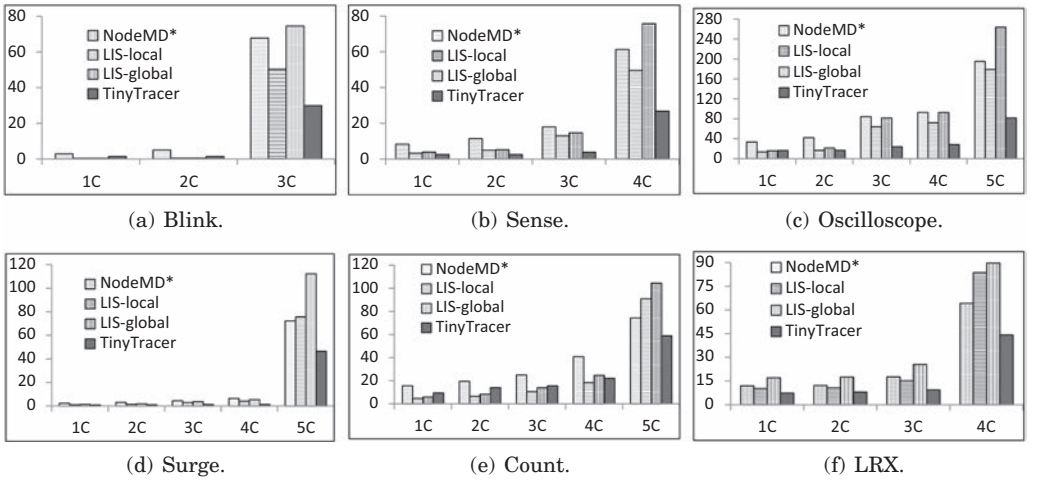
Since the overhead of tracing depends on the number of traced nesC components, we traced all the important nesC components, including the application component (i.e., `SurgeM` for Surge), as well as the system components, such as for LEDs (i.e., `LedsC`), sensing (i.e., `PhotoTempM`), single-hop network communication (i.e., `AMStandard`) or multihop network communication (i.e., `MultihopEngineM`), and timer (i.e., `TimerM`). The order in which the components were included in the tracing simulation is the following; application component, LED, sensor, network layer, and timer. Every benchmark has at least the first two components and one or more of the other three. The nesC component here corresponds to a layer of functionality in TinyOS. Each component corresponds to many C functions, and tracing a nesC component implies tracing all the corresponding C functions. Table III shows a mapping from number of components traced to the number of C functions traced for every benchmark. An example for interpreting table III is as follows. The number 14 in the third row third column means that the two components in benchmark Sense has 14 C functions.

6.2. State-of-the-Art Techniques

We compare TinyTracer against two state-of-the-art WSN tracing techniques, namely, NodeMD [Krunic et al. 2007] and LIS [Shea et al. 2010]. In terms of trace size, we also compare TinyTracer to the MR algorithm, a state-of-the-art interprocedural control flow path encoding algorithm for traditional systems.

NodeMD records a trace of function calls and thread identifiers but does not record control flow information. For fair comparison, we added intraprocedural control flow tracing based on the BL algorithm to NodeMD. We refer to this combination as *NodeMD**.

LIS (Log Instrumentation Specification) provides a language for describing runtime data of interest, as well as a runtime environment for automatically collecting it efficiently. LIS supports tracing control flow of arbitrary sets of functions. The key ideas



(a) Blink. (b) Sense. (c) Oscilloscope.

(d) Surge. (e) Count. (f) LRX.

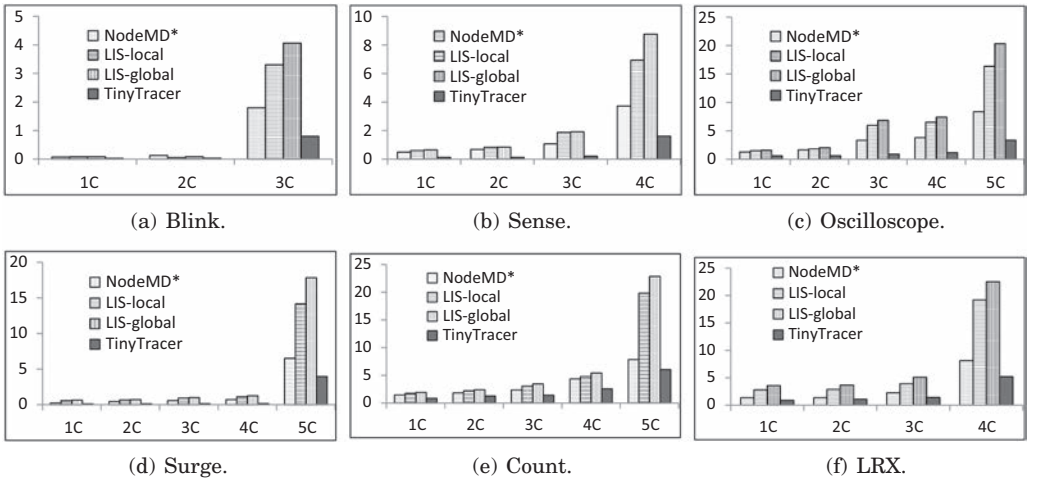
Fig. 15. Total energy overhead of control flow tracing using various techniques as a percentage of total energy used by baseline (no tracing).

of LIS are bit-aligned logging or variable bit encoding and locally scoped identifiers for reducing trace size and resource consumption. We ported it from TinyOS 2.x to TinyOS 1.x. Since the traces generated with local identifiers for conditional statements do not always parse correctly with the parser included in the distribution (before the porting), we also tested the configuration for global identifers, which did not reveal similar issues. We refer to experiments run with these two options as *LIS-local* and *LIS-global* respectively. LIS refers to either the approach itself or either one of *LIS-local* or *LIS-global* when they behave identically.

6.3. Energy Overhead

For each benchmark, we measured the energy consumption of NodeMD*, LIS, and TinyTracer for different components being traced. The energy overhead due to tracing is caused by CPU usage and the external flash. The total and CPU energy overheads are shown in Figures 15 and 16 respectively. "C" on the x-axes of these figures refers to the number of nesC components included in tracing. The number of C functions corresponding to those components is shown in Table III. The flash energy overhead can be computed from these graphs. The original implementations of LIS and NodeMD did not expect long executions, and thus, they may generate traces that cause flash overflow. Therefore, we ran those benchmarks for 7.5 minutes and scaled it to one hour to compare against TinyTracer.

We observe from Figure 15 that TinyTracer's energy overhead is low (0.88% for Surge to 9.43% for Count) when a single application component is being traced. As we increase the number of components traced, the overhead increases. However, it is interesting to note that even when including most system components—LEDs, sensors, and network layer (except timer)—the overhead did not increase (1.33% for Surge to 28.36% for Oscilloscope) substantially, even for LRX. In addition to the number of components traced, tracing overhead directly depends on how often the node is awake. Hence, Oscilloscope and Count have more overhead than LRX or Surge, in which the nodes are mostly asleep. Real-world sensing applications that are awake only once every few minutes would generate similarly low overhead. When the timer is included, the overhead increases noticeably (ranging from 26.78% for Sense with 4C to 81.76% for Oscilloscope with 5C). This is due to the frequent execution of parts of



(a) Blink. (b) Sense. (c) Oscilloscope.

(d) Surge. (e) Count. (f) LRX.

Fig. 16. CPU energy overhead of control flow tracing using various techniques as a percentage of CPU energy used by baseline (no tracing).

code that interact with the hardware (e.g., responding to frequent clock interrupts), as well as the presence of multiple loops. From the results, we infer that our control flow tracing indeed incurs low overhead for most of the application and system components.

We observe from Figure 15 that the total energy overhead due to TinyTracer (0.9% to 81.8%) is comparable or (significantly) lower compared to that of NodeMD* (2.4% to 195.3%), LIS-local (0.4% to 179.1%), and LIS-global (0.8% to 263.65%), depending on the number of components being traced. For simple cases, such as Blink, or when tracing only a few components, LIS-local performs comparably or better due to variable bit encoding used by LIS-local. We illustrate in Section 6.5.1 that TinyTracer augmented with variable bit encoding reduces trace size more than LIS.

When tracing more components or complex programs, TinyTracer incurs significantly lower energy overhead than NodeMD* (up to 79.1% lower), LIS-local (up to 70.9%), and LIS-global (up to 75.5%). The two main reasons being the encoding of interprocedural paths and compression. Since many functions in TinyOS are small and do not contain loops, such functions can be succinctly represented using interprocedural path identifier. However, NodeMD* and LIS record function identifiers and the control flow paths for each of these functions separately. Furthermore, the succinct interprocedural path encoding enables opportunities for compression and simple compression techniques used by TinyTracer compresses away the repeating patterns.

We observe from Figure 16 that the CPU energy overhead for TinyTracer is very low (0.03% to 5.2%). From this, we observe that the flash energy overhead dominates the total energy overhead. The flash energy overhead is directly related to the trace size. Trace compression used by TinyTracer reduces the trace size and thus the flash energy overhead.

LIS and NodeMD* incur significantly more CPU overhead than TinyTracer for all benchmarks (up to 9× and 5×, respectively). The main reason being the inexpensive instrumentation used by TinyTracer. At every branch, TinyTracer only performs an increment, as opposed to a function call, in LIS. Due to interprocedural path recording, fewer function calls are required to record procedure entries and exits compared to NodeMD*.

Table IV. Trace Size for 30-Minute Runs

Benchmark	Compressed Trace Size (B)	Compression Ratio	Number of Traced Functions
Blink	344	21.71	5
Sense	3,040	9.6	7
Oscilloscope	4,864	11.99	11
Surge	568	14.11	11
Count	3,664	4.625	6
LRX	93,200	1.74	29

6.4. Trace Compression

We show that our compression scheme is effective and generates smaller traces. The trace sizes and compression ratios recorded for each of the benchmarks are shown in Table IV. The data compression ratio is the ratio of uncompressed size to compressed size. We observe that the amount of trace information generated in 30 minutes for a normal WSN application is less than a one KB (568 for Surge) and for a high-throughput applications, is about a few KBs (4,864 for Oscilloscope). For a complex application like LRX that contains multiple loops and several function calls, the trace size is still manageable and can be reduced further by tracing only important parts of the application.

6.5. Trace Size Comparison

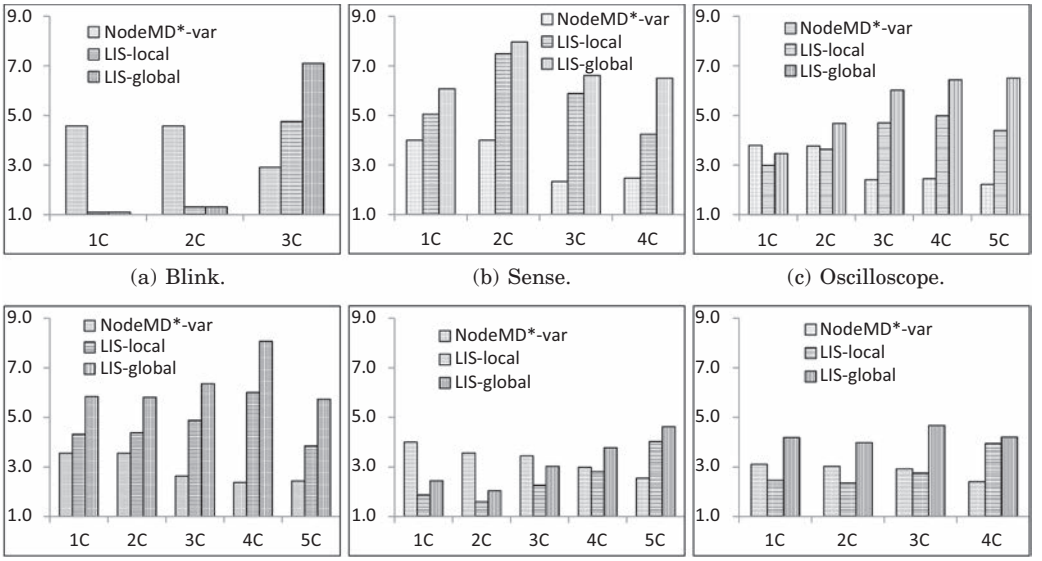
We compare TinyTracer with the state-of-the-art tracing or profiling approaches in terms of trace sizes.

6.5.1. Variable Bit Encoding. TinyTracer uses fixed bit encoding for the trace entries. In our experiments, we used two bytes to encode the trace entry to cover for the worst case. However, most of the trace entries require fewer bits to encode. For example, a function with n paths can be recorded using $(\log_2 n) + 2$ bits. Out of the two additional bits, one is used to distinguish a function identifier from a path entry and the other is used as an ***end*** symbol. We call such an encoding *variable bit encoding* of trace entries. Such an encoding can further reduce the trace size. In this section, we compare various tracing techniques using variable bit encoding.

LIS already uses variable bit encoding. To compare the variable bit encoded traces, generated by LIS, TinyTracer, and NodeMD*, we created *TinyTracer-var* and *NodeMD*-var* as follows. For fairness, we disabled the compression in TinyTracer and collected uncompressed traces generated by TinyTracer. We analyzed the program to identify the required number of bits to encode each trace entry in the uncompressed trace. We reinterpreted the uncompressed trace collected using variable bit encoding to obtain the size of TinyTracer-var. We obtained NodeMD*-var analogously. We ran all the techniques on ATEMU for 7.5 minutes and collected the uncompressed traces from flash.

Figure 17 shows the comparison of trace sizes generated by NodeMD*-var, LIS-local, and LIS-global as a factor of trace size generated by TinyTracer-var. NodeMD*-var, LIS-local, and LIS-global yield traces which are 10% to 810% larger than those of TinyTracer-var: 2.2× to 4.6× more for NodeMD*-var, 1.1× to 7.5× more for LIS-local, and 1.1× to 8.1× more for LIS-global. The reason for smaller trace sizes for TinyTracer-var being the succinct interprocedural path encoding, which reduces the number of trace entries in TinyTracer-var.

6.5.2. Comparison with MR Algorithm. We now compare the size of the traces generated by our algorithm with the MR algorithm and show that our algorithm is comparable in



(a) Blink. (b) Sense. (c) Oscilloscope.

(d) Surge. (e) Count. (f) LRX.

Fig. 17. Sizes of variable bit encoded traces generated by various tracing techniques as a factor of sizes generated by TinyTracer.

size or better. The MR algorithm first has to be adapted to the concurrent environment similar to our algorithm, that is, applying it to every execution unit (task or event). Since there is no open implementation available of the MR algorithm (i.e., there are no experimental results in the original paper [Melski and Reps 1999]), 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 (i.e., functions other than tasks or events) containing loops. Both algorithms record the loop paths for every iteration of a loop. In the MR algorithm, loop paths use globally unique path identifiers, thus increasing the global identifiers, space significantly; in our algorithm, however, loop paths use locally unique identifier, and the function context is used 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 Figure 18. The columns in the table correspond to the size of the traces generated by the MR algorithm normalized by size of the respective traces 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 one bit more, because there are only few non

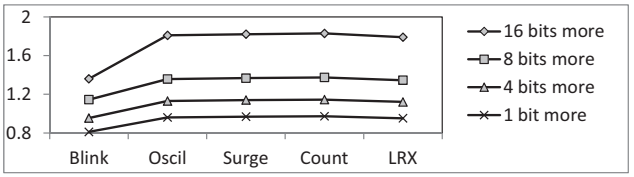


Fig. 18. Trace size comparison with the MR algorithm.

top-level functions with loops in the benchmark programs. When the global identifier uses one byte or two bytes more, our algorithm saves considerable space (14% to 83%). Thus, as opposed to a direct adaptation of the MR algorithm, our algorithm is better for WSNs because it uses fewer CPU cycles per instrumentation, generates smaller traces, and is much simpler to implement.

6.6. Memory Consumption

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 along with `-finline-limit=100000`.

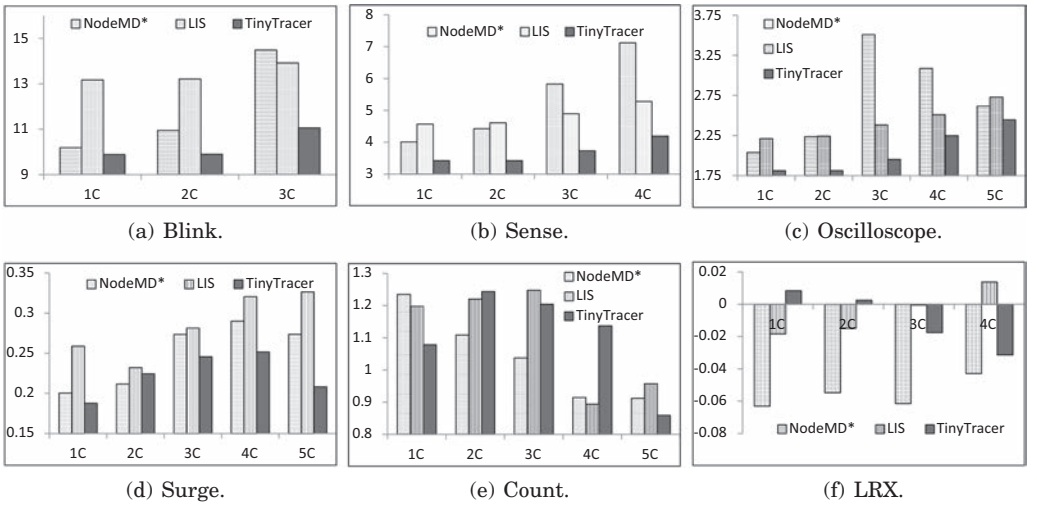
We note that there are two ingredients to program memory overhead: *fixed* and *variable*. The former is due to the internal algorithms, such as the generic compression used by our tracing component and the device driver code for the flash component. The latter ingredient corresponds to instrumentation. The program memory overhead for NodeMD*, LIS, and TinyTracer as a factor of baseline program memory is shown in Figure 19. For simple programs like Blink and Oscilloscope, the overhead is high because of the fixed components that need to be added. But for large programs, such as Surge and LRX, the fixed component increase is less 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. Note that the effect of removal of inlining on execution time is low, 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 the mica (mica2, micaz) family or 48 KB for the telosb families of motes. Since the nesC compiler suggests aggressive inlining of the program, reducing inlining could help large programs. TinyTracer uses less program memory than LIS and NodeMD* for most benchmarks. For Count and LRX, NodeMD* and LIS use less program memory when tracing fewer components. We attribute these variations to function inlining.

The data memory requirement is around 950 B for TinyTracer and NodeMD* and about 450 B for LIS. This is usually independent of the instrumented program. The major contributor to the data memory is the internal buffers 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. For most programs, half of that size is enough. The split of the data memory is shown in Table V.

7. DISCUSSION

Our approach is a first step to creating a postmortem 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 aid in diagnosis of a large class of errors that change control flow including logical errors, high-level



(a) Blink. (b) Sense. (c) Oscilloscope.

(d) Surge. (e) Count. (f) LRX.

Fig. 19. Program memory overhead for NodeMD*, LIS, and TinyTracer as a factor of program memory usage of baseline (no tracing).

Table V. Division of Data Memory Overhead

Data Structure	TinyTracer	LIS
Buffer used by Flash	256B	336B
Circular Trace Buffer	384B	21B
Miscellaneous	$\tilde{3}$00B	$\tilde{7}$0B

race conditions, node reboots, network failures (i.e., node/link failures), and memory errors with varying overhead. However, specific approaches could be less expensive for the respective targeted classes of errors, such as Safe TinyOS [Cooprider et al. 2007], for memory-specific errors, or could require less manual intervention such as Sympathy [Ramanathan et al. 2005] or PAD [Liu et al. 2008] 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 postmortem analysis, the developer would not be able to localize the fault with the control flow trace alone. We note that during postmortem analysis, tools such as Chess [Musuvathi et al. 2008] 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 also prevents most low-level race conditions.

Another limitation of any tracing mechanism is that these are intrusive and may cause timing-dependent faults to disappear when tracing is turned on. We minimize the intrusive effect by scheduling the time-consuming operations, such as compression or writing to flash, when the system is not doing timing-critical operations such as interrupt handling. This is achieved by handling these time-consuming operations with TinyOS tasks that yield frequently.

We observe that the trace size can increase when many components, especially system components such as timers are traced. The functions to trace depend on the developer's confidence in his code. It is usually the case that the developer wants to trace the application components that are more likely to contain errors rather than well-tested system components, which are stable. As a guideline, tracing all the application components is a good starting point, and as the code matures, tracing only the execution units (event handlers and tasks) of application components is helpful because subtle interactions of these execution units cause several complex bugs to be reported, including the ones in Section 5.3. When the tracing of system components is required, it is beneficial to trace the parts of system components that interact with the application components before tracing the internal parts. The internal parts of system components interact with the hardware (such as sending one bit on the radio or checking the clock to fire a timer) and execute frequently, resulting in large trace size.

8. RELATED WORK

Debugging WSNs has been an important research problem for several years, and many interesting solutions have been proposed. Existing work in debugging for WSNs can be classified into (1) tools that are used in pre-deployment testing, (2) tools that provide insight into the network, (3) tools that help in automating debugging, and (4) tools that trace the execution.

8.1. Testing

During development, WSN applications are usually tested and debugged using simulators, or testbeds. WSN-specific simulators, such as TOSSIM [Levis et al. 2003], or emulators, such as ATEMU [Polley et al. 2004] and Avrora [Titzer et al. 2005], are invaluable tools for checking functionality in the early development stages. Since simulation/emulation trade fidelity for scalability, testbeds are usually used to gain confidence in the implementation. Large-scale public testbeds, such as Motelab [Werner-Allen et al. 2005] and Kansei [Ertin et al. 2006], have been very useful for testing WSN applications.

Several program analysis-based testing tools have also been developed for WSNs. An application graph abstraction to represent nesC programs was explored in Nguyen and Soffa [2007]. The abstraction, however, misses the interrupt resumption edges. Intercontext CFG and DFG [Lai et al. 2008] use a complete application graph that represents all possible edges, leading to a very large graph. Such a graph is used to mine test cases. Kothari et al. [2008] use symbolic execution to extract the finite state machine implied by the TinyOS code, which aids in better understanding of the code. Li and Regehr [2010] use model checking to find safety and liveness errors in TinyOS applications running on TOSSIM. While these tools are helpful in development and testing, they are not meant for runtime debugging.

8.2. 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. Marionette [Whitehouse et al. 2006] is 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. Clairvoyant [Yang et al. 2007] 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. TinyLTS [Sauter et al. 2011] provides an efficient `printf` mechanism that allows for printing or sending some state information after deployment to the base station. Tolle and Culler [2005] proposed SNMS,

a management system for collecting various information about the network, using a separate radio stack to send the logs. These visibility tools can incur large runtime overhead, as they require several message exchanges before the root causes can be found. In contrast, by recording traces and doing postmortem diagnosis, we can reduce this runtime overhead significantly.

8.3. Automated Debugging

Several approaches have been proposed for monitoring the network for error detection and problem diagnosis in WSNs. Sympathy [Ramanathan et al. 2005] is perhaps the first network diagnosis approach specifically designed for WSNs. Sympathy 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. PAD [Liu et al. 2008] is similar to Sympathy but uses bayesian network-based analysis to reduce network monitoring traffic and infer network failures and their causes. Livenet [Chen et al. 2008] and SNTS [Khan et al. 2007] monitor messages using special nodes that require extra hardware. These solutions focus on network faults and node and link failures but do not handle application implementation faults.

Envirolog [Luo et al. 2006] records all the events in a given time period and replays them within the node at a later point in time. This solution is proposed for infield parameter tuning, and the high overhead incurred due to the tracing of all events and their input values is acceptable for the intended use. Herbert et al. [2007] present an invariant-based, application-level runtime monitoring tool that can be used to detect errors in WSN applications. Hermes [Kothari et al. 2008] is similar to HSEND [Herbert et al. 2007] but allows developer to write invariants dynamically as well as to push patches that could fix the violation at runtime. These approaches help monitor WSNs at a higher granularity, and the error detection by these approaches complement diagnostic tracing. However, these approaches expect the user to write invariants, which may be difficult in some cases, since writing these invariants requires some knowledge about the failures beforehand.

Dustminer [Khan et al. 2008] collects event logs from both good and bad executions and uses machine learning techniques to identify discriminative frequent patterns, which can aid in diagnosis. The technique proposed can be used with control flow traces instead of event logs. One disadvantage of machine learning approaches is that they require multiple runs to learn.

8.4. Tracing

There have been a few trace-based approaches proposed for diagnosing faults in WSNs. Declarative TracePoints [Cao et al. 2008] provides a uniform SQL-based query language interface for debugging and can simulate other trace-based approaches, such as the ones described next. NodeMD [Krunic et al. 2007] records system calls and context switches to detect stack overflows, livelocks, 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 is useful for specific problems, such as stack overflow, but not for general application errors. It does not contain the exact control flow path information, making it difficult to reproduce faults. LIS [Shea et al. 2010] proposes a log instrumentation specification language and runtime, which provides a systematic way to describe and collect information about the execution, such as function calls and control flow paths executed in a WSN. LIS is optimized to collect function calls as well as control flow trace efficiently. However, as observed in Section 6, LIS incurs more overhead than TinyTracer due to the lack of interprocedural control flow path encoding and compression. Furthermore, due to tight bit alignment, it is

susceptible to synchronization errors if there is packet loss or bit corruption unless redundant bits are added. In contrast, our approach produces succinct traces because interprocedural control flow paths contain information about several function calls and intraprocedural paths.

In traditional systems, bit tracing [Ball and Larus 1994] has been proposed for tracing the control flow by recording one bit for every predicate. The disadvantage of this approach is that, as mentioned in [Ball and Larus 1996], such a tracing is not optimal in terms of encoding space 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.

The MR algorithm proposed by Melski and Reps [1999] is an interprocedural control flow path profiling algorithm. Unlike BL algorithm, edge increments created by the MR algorithm are linear functions involving multiplications to handle context-sensitivity. In comparison, edge increments in our approach are simple additions, except at function call sites where one multiplication is used. Having fewer expensive instructions is particularly important for sensor programs. Furthermore, our approach takes advantage of local identifiers for repeating loop paths, thus saving on the number of bits required to encode the paths and reducing the number of CPU cycles required to calculate the identifiers. We showed a quantitative comparison in Section 6.5.2 which showed significant reduction in the trace size. Furthermore, our algorithm is much simpler to implement because our interprocedural summary analysis is modular. In contrast, the MR algorithm is very complex.

9. CONCLUSIONS AND FUTURE WORK

In this article, we showed that program tracing can be performed efficiently and accurately in WSNs. To this end, we introduced a novel approach for efficient intraprocedural and interprocedural control flow tracing and encoding for WSNs. With the help of real-world case studies and applications, we showed that such fine-grained traces are very useful in postmortem diagnosis of complex faults in WSNs.

Backed by this evidence, we are exploring various improvements to WSN tracing, such as extending our scheme to trace messages efficiently as well as novel trace compression methods suitable for WSNs. Furthermore, we are developing a replay tool that can aid developers with quicker diagnosis of faults.

REFERENCES

Ball, T. and Larus, J. R. 1994. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst. 16*, 1319–1360.

Ball, T. and Larus, J. R. 1996. Efficient path profiling. In *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarcritecture (MICRO)*.

Beutel, J., Römer, K., Ringwald, M., and Woehrle, M. 2009. Deployment techniques for sensor networks. In *Sensor Networks: Where Theory Meets Practice,* G. Ferrari, Ed., Signals and Communications Theory, Springer Verlag, Berlin Heiddberg, 219–248.

Cao, Q., Abdelzaher, T., Stankovic, J., Whitehouse, K., and Luo, L. 2008. Declarative tracepoints: A programmable and application independent debugging system for wireless sensor networks. In *Proceedings of the ACM Conference on Embedded Sensor Networks (SenSys)*.

Chen, B., Peterson, G., Mainland, G., and Welsh, M. 2008. LiveNet: Using passive monitoring to reconstruct sensor network dynamics. In *Proceedings of the IEEE International Conference on Distributed Computing in Sensor System (DCOSS)*.

Cooprider, N., Archer, W., Eide, E., Gay, D., and Regehr, J. 2007. Efficient memory safety for TinyOS. In *Proceedings of the ACM Conference Embedded Sensor Networks (SenSys)*.

Dunkels, A., Grönvall, B., and Voigt, T. 2004. Contiki - A lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN'04)*.455–462.

DUNKELS, A., SCHMIDT, O., VOIGT, T., AND ALI, M. 2006. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the ACM Conference Embedded Sensor Networks (SenSys)*.

ERTIN, E., ARORA, A., RAMNATH, R., NAIK, V., BAPAT, S., KULATHUMANI, V., SRIDHARAN, M., ZHANG, H., CAO, H., AND NESTERENKO, M. 2006. Kansei: A testbed for sensing at scale. In *Proceedings of the ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*.

GEELS, D., ALTEKAR, G., MANIATIS, P., AND ROSCOE, T. 2007. Friday: Global comprehension for distributed replay. In *Proceedings of the ACM Symposium on Networked Systems Design and Implementation (NSDI)*.

HASLER, A., TALZI, I., BEUTEL, J., TSCHUDIN, C., AND GRUBER, S. 2008. Wireless sensor networks in permafrost research - concept, requirements, implementation and challenges. In *Proceedings of the 9th International Conference on Permafrost (NICOP)*.

HEINZELMAN, W. B., CHANDRAKASAN, A. P., AND BALAKRISHNAN, H. 2002. An application-specific protocol architecture for wireless microsensor networks. *IEEE Trans. Wirel. Commun. 1,* 4, 660–670.

HERBERT, D., SUNDARAM, V., LU, Y. H., BAGCHI, S., AND LI, Z. 2007. Adaptive correctness monitoring for wireless sensor networks using hierarchical distributed run-time invariant checking. *ACM Trans. Auton. Adapt. Syst. 2,* 3, 8.

KELLER, M., BEUTEL, J., MEIER, A., LIM, R., AND THIELE, L. 2009. Learning from sensor network data. In *Proceedings of the ACM Conference on Embedded Sensor Networks (SenSys)*.

KHAN, M. M. H., LE, H., AHMADI, H., ABDELZAHER, T., AND HAN, J. 2008. Dustminer: Troubleshooting interactive complexity bugs in sensor networks. In *Proceedings of the ACM Conference on Embedded Sensor Networks (SenSys)*.

KHAN, M. M. H., LUO, L., HUANG, C., AND ABDELZAHER, T. F. 2007. SNTS: Sensor network troubleshooting suite. In *Proceedings of the IEEE International Conference on Distributed Computing in Sensor System (DCOSS)*.

KLUES, K., LIANG, C.-J. M., PAEK, J., MUSALOIU-ELEFTERI, R., LEVIS, P., TERZIS, A., AND GOVINDAN, R. 2009. TOSThreads: Thread-safe and non-invasive preemption in TinyOS. In *Proceedings of the ACM Conference on Embedded Sensor Networks (SenSys)*.

KOTHARI, N., MILLSTEIN, T., AND GOVINDAN, R. 2008. Deriving state machines from TinyOS programs using symbolic execution. In *Proceedings of the ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*.

KOTHARI, N., NAGARAJA, K., RAGHUNATHAN, V., SULTAN, F., AND CHAKRADHAR, S. 2008. Hermes: A software architecture for visibility and control in wireless sensor network deployments. In *Proceedings of the ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*.

KRUNIC, V., TRUMPLER, E., AND HAN, R. 2007. NodeMD: Diagnosing node-level faults in remote wireless sensor systems. In *Proceedings of the ACM International Conference on Mobile Systems, Analysis and Simulation of Wireless Mobile Systems (MobiSys)*.

LAI, Z., CHEUNG, S. C., AND CHAN, W. K. 2008. Inter-context control-flow and data-flow test adequacy criteria for nesC applications. In *Proceedings of the ACM International Symposium on Foundations of Software Engineering (FSE)*.

LARUS, J. R. 1999. Whole program paths. In *Proceedings of the ACM Conference on Programming Language Design and Implementation (PLDI)*.

LEVIS, P., LEE, N., WELSH, M., AND CULLER, D. 2003. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *Proceedings of the ACM Conference on Embedded Sensor Networks (SenSys)*.

LEVIS, P., MADDEN, S., POLASTRE, J., SZEWCZYK, R., WHITEHOUSE, K., WOO, A., GAY, D., HILL, J., WELSH, M., BREWER, E., AND CULLER, D. 2005. TinyOS: An operating system for sensor networks. In *Ambient Intelligence*, W. Weber, J. M. Rabaey, and E. Aarts, Eds. Springer Verlag, Berlin Heidelberg Chapter 7.

LI, P. AND REGEHR, J. 2010. T-Check: Bug finding for sensor networks. In *Proceedings of the ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*.

LIU, K., LI, M., LIU, Y., LI, M., GUO, Z., AND HONG, F. 2008. PAD: Passive diagnosis for wireless sensor networks. In *Proceedings of the ACM Conference on Embedded Sensor Networks (SenSys)*.

LIU, X., GUO, Z., WANG, X., CHEN, F., LIAN, X., TANG, J., WU, M., KAASHOEK, M. F., AND ZHANG, Z. 2008. D3S: Debugging deployed distributed systems. In *Proceedings of the ACM Symposium on Networked Systems Design and Implementation (NSDI)*.

LUO, L., HE, T., ZHOU, G., GU, L., ABDELZAHER, T. F., AND STANKOVIC, J. A. 2006. Achieving repeatability of asynchronous events in wireless sensor networks with EnviroLog. In *Proceedings of the Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*.

MELSKI, D. AND REPS, T. W. 1999. Interprocedural path profiling. In *Proceedings of the ACM International Conference on Compiler Construction (CC)*.

MUSUVATHI, M., QADEER, S., BALL, T., BASLER, G., NAINAR, P. A., AND NEAMTIU, I. 2008. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the ACM USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

NGUYEN, N. T. M. AND SOFFA, M. L. 2007. Program representations for testing wireless sensor network applications. In *Proceedings of the Workshop on Domain-Specific Approches to Software Test Automation in conjuction with the 6th ESEC/FSE Joint Meeting (DOSTA'07)*. 20–26.

POLLEY, J., BLAZAKIS, D., MCGEE, J., RUSK, D., AND BARAS, J. S. 2004. ATEMU: A fine-grained sensor network simulator. In *Proceedings of the IEEE International Conference on Sensor and Ad Hoc Communications and Networks (SECON)*.

RAMANATHAN, N., CHANG, K., KAPUR, R., GIROD, L., KOHLER, E., AND ESTRIN, D. 2005. Sympathy for the sensor network debugger. In *Proceedings of the ACM Conference on Embedded Sensor Networks (SenSys)*.

SAUTER, R., SAUKH, O., FRIETSCH, O., AND MARRÓN, P. J. 2011. TinyLTS: Efficient network-wide logging and tracing system for TinyOS. In *Proceedings of the IEEE Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*.

SHEA, R., SRIVASTAVA, M., AND CHO, Y. 2010. Scoped identifiers for efficient bit aligned logging. In *Proceedings of the Conference and Exhibition on Design, Automation and Test in Europe (DATE)*.

SOOKOOR, T., HNAT, T., HOOIMEIJER, P., WEIMER, W., AND WHITEHOUSE, K. 2009. Macrodebugging: Global views of distributed program execution. In *Proceedings of the ACM Conference on Embedded Sensor Networks (SenSys)*.

SUNDARAM, V., BAGCHI, S., LU, Y.-H., AND LI, Z. 2008. SeNDORComm: An energy-efficient priority-driven communication layer for reliable wireless sensor networks. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems (SRDS)*.

SUNDARAM, V., EUGSTER, P., AND ZHANG, X. 2010. Efficient diagnostic tracing for wireless sensor networks. In *Proceedings of the ACM Conference on Embedded Sensor Networks (SenSys)*.

SUNDARAM, V., EUGSTER, P., AND ZHANG, X. 2011. Demo Abstract: Diagnostic tracing of wireless sensor networks with TinyTracer. In *Proceedings of the ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*.

TITZER, B. L., LEE, D. K., AND PALSBERG, J. 2005. Avrora: Scalable sensor network simulation with precise timing. In *Proceedings of the ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*.

TOLLE, G. AND CULLER, D. 2005. Design of an application-cooperative management system for wireless sensor networks. In *Proceedings of the 2nd European Workshop on Wireless Sensor Networks (EWSN)*.

WELCH, T. A. 1984. Technique for high-performance data compression. *Computer 17,* 6.

WERNER-ALLEN, G., SWIESKOWSKI, P., AND WELSH, M. 2005. MoteLab: A wireless sensor network testbed. In *Proceedings of the ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*.

WHITEHOUSE, K., TOLLE, G., TANEJA, J., SHARP, C., KIM, S., JEONG, J., HUI, J., DUTTA, P., AND CULLER, D. 2006. Marionette: Using RPC for interactive development and debugging of wireless embedded networks. In *Proceedings of the ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*.

YANG, J., SOFFA, M. L., SELAVO, L., AND WHITEHOUSE, K. 2007. Clairvoyant: A comprehensive source-level debugger for wireless sensor networks. In *Proceedings of the ACM Conference on Embedded Sensor Networks (SenSys)*.

Received December 2011; revised June 2012; accepted July 2012