Lazy Preemption to Enable Path-Based Analysis of Interrupt-Driven Code

Wei Le, Jing Yang, Mary Lou Soffa and Kamin Whitehouse
Department of Computer Science
University of Virginia
Charlottesville, Virginia 22904
{weile,jy8y,soffa,whitehouse}@virginia.edu

ABSTRACT

One of the important factors in ensuring the correct functionality of wireless sensor networks (WSNs) is the reliability of the software running on individual sensor nodes. Research has shown that path-sensitive static analysis is effective for bug detection and fault diagnosis; however, path-sensitive analysis is prohibitively expensive when applied to a WSN application due to the large state space caused by arbitrary interrupt preemptions. In this paper, we propose a new execution model called *lazy preemption* that reduces this state space by restricting interrupt handlers to a set of predetermined *preemption points*, if possible. This execution model allows us to represent the program with an *inter-interrupt control flow graph (IICFG)*, which is easier to analyze than the original CFGs with arbitrary interrupt preemptions.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*Reliability*

General Terms

Reliability, Algorithms

Keywords

Lazy preemption, Path-based analysis, Interrupt-driven

1. INTRODUCTION

Reliability is important for wireless sensor networks (WSNs), especially when they are deployed in mission-critical applications such as security monitoring or power plant control. Software reliability is particularly challenging because WSN programs often use interrupt-driven code, which is difficult for developers to reason about and is highly susceptible to race conditions and other bugs [13]. To make matters worse, traditional tools and techniques for software reliability are handicapped in the WSN domain. For example, resource constraints limit the use of debuggers, virtualization, memory sandboxing, and other runtime protections. Furthermore, sensor nodes are often deployed in remote locations where continuous debugging and reprogramming is not an option. Finally, testing and debugging are limited by the large range of possible input sequences, and real deployment environments are difficult to emulate in a testing environment.

Static analysis can improve software reliability without paying a runtime cost because it is performed before the code is deployed. However, WSN programs are especially difficult to analyze because they are interrupt-driven. Preemptive interrupts allow the code to be highly responsive to the environment, but exponentially increase the number of possible paths through a program that must be analyzed: if k different interrupt handlers can be run at n different points along a path then k^n new variants of the path must be considered during program analysis. Even more variants must be considered if interrupts can preempt other interrupts. Thus, it is intractable for path-based analysis to achieve reasonable coverage of interrupt-driven code, even for very small programs. Existing static analysis tools designed for the WSN domain address this challenge by either 1) sacrificing code coverage by analyzing only a limited number of paths [12], or 2) sacrificing accuracy by using path-insensitive analysis that can produce a large number of false positives or false negatives [2, 3, 11].

In this paper, we propose to address this problem with *lazy preemption*, a new execution model that runs interrupt handlers only when necessary or convenient. This execution model reduces the number of interleavings caused by asynchronous interrupt preemption, thereby simplifying path-based analysis. To use lazy preemption, a maximum frequency f_i and maximum response time r_i must be defined for every hardware interrupt i. The default values are $f_i = 500$ Hz (the interrupt occurs at most twice every 1 msec) and $r_i = 1$ msec (the handler must be executed within 1 msec of the hardware interrupt). Then, we split all interrupt handlers into two parts: the *record handler* and the *action handler*. The record handler executes immediately when a hardware interrupt occurs and can record the state of the hardware, but cannot share variables with any code other than its action handler. The action handler processes the state and updates shared variables, but it can only execute at pre-determined *preemption points*, which are chosen at compile time based on static timing analysis such that timing constraints are satisfied. This lazy preemption model enables a new intermediate representation of the program called the *Inter-Interrupt Control Flow Graph (IICFG)* that models interleavings of action handlers only at their correspondent preemption points. Record handlers do not need to be modeled because they do not use shared variables. The IICFG is easier to analyze than the original CFGs with arbitrary interrupt preemption because it represents far fewer possible program paths.

Lazy preemption creates a new tradeoff between software verifiability and software responsiveness, and allows a user to balance

SESENA '11, May 22, 2011, Waikiki, Honolulu, HI, USA

the two by specifying the frequency and response time constraints accordingly; if a high maximum frequence and low maximum response time are specified, the program will be more responsive at run time but analysis will take longer at compile time, and vice versa. In this paper, we argue that a large fraction of WSN programs would benefit by reducing responsiveness and improving verifiability. The key insight is that the physical world changes much more slowly (on the order of seconds) than the time required to execute typical software routines (on the order of milliseconds). For example, low-power WSNs deliberately turn down the sensing frequency to increase battery lifetime, sensing every few seconds or even every few minutes. Even timing-critical systems that detect fires or heart attacks can tolerate a second or two of latency. In comparison, we found that 75% of tasks and interrupt handlers in TinyOS-1.x have less than 1000 lines of code, and most can execute in less than a millisecond. Thus, it should be possible to substantially reduce software complexity caused by preemption simply by postponing interrupt handler routines, without paying a significant penalty in terms of responsiveness.

Of course, some interrupts need to be handled immediately and some tasks can run for seconds or even minutes. We envision three types of lazy preemption: 1) *fully-preemptive*: the interrupt must be handled immediately, 2) *restricted-preemptive*: the maximum response time is less than the time required to execute certain software routines in the program, and fixed preemption points must be defined, and 3) *non-preemptive*: the maximum response time is large enough with respect to software execution time that the interrupt handler can be queued and run whenever the stack is empty. In the latter case, all subroutines are essentially atomic and static analysis is greatly simplified. We envision that each program will have a combination of these preemption models, perhaps one for each interrupt.

We have developed an initial prototype that uses demand-driven, path-based static analysis on the IICFG built based on the non-preemptive model. Demand-driven fault detection starts at program points where a fault potentially occurs and examines the program paths reachable from these program points [6, 7]. The prototype is able to detect buffer overflows, integer faults and null-pointer dereferences. We are still implementing the static timing analysis and the dynamic monitor.

2. BACKGROUND

Our techniques presented in this paper model software running on individual sensor nodes, rather than the behavior of an entire wireless sensor network. To explain the techniques, we use TinyOS as an example which compiles programs written in nesC to C code; for hardware-dependent features, we use the AVR platform. Our techniques are generally applicable for WSN applications written in other languages and can be adapted to other hardware platforms.

In this section, we present the execution model of WSN applications. The focus is to explain the characteristics of WSN applications that are not commonly seen in traditional software. We also discuss the types of faults that have been reported for WSN applications. The goal is to determine the requirements for static analysis that is able to detect these faults.

2.1 The Execution Model

Compared to traditional software, a WSN application is typically smaller, ranging from hundreds to 20 k lines of C code [3]. Three major components in a WSN application include the *main* function, *tasks* and *interrupt handlers*, as shown in Figure 1. The major role of the main function is to fetch tasks from the task queue and execute them in FIFO (see the loop in the main function). A *task*

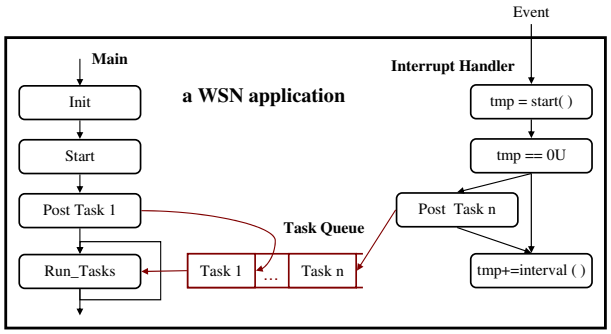

Figure 1: Execution model of a WSN application.

is a basic computational unit that accomplishes a desired functionality. Tasks are implemented as function calls in C, using special naming conventions to distinguish them from other function calls. Tasks can be posted by the main function, by interrupt handlers or by other tasks. The interrupt handlers are invoked to respond external signals. In most WSN applications, the signals come at a fixed frequency. The frequency is potentially known from the hardware specification or the documentation. To process the signal, tasks are invoked and placed in the task queue. There are often global variables in the interrupt handlers that can record the accumulated status for the captured signals.

At any given time, the CPU executes instructions from either the main function, a task or an interrupt handler. An interrupt handler can preempt a task or another interrupt handler. When the execution is not declared as atomic, the system examines the interrupt vector table at each instruction and processes the cached interrupts based on their priority. The table can only store one interrupt of a type at one time. When the task and the interrupt handler are preempted, the program state is pushed onto the stack and resumed after the preempting interrupt handlers terminate.

For some WSN applications, interrupts must be handled in a timely manner. The real-time constraints are often specified in terms of response time for external events. The arrival frequency of an interrupt (f) and its response time (r) are related. Since the system is not able to record more than one interrupt of the same type at a time, for a system where missing an interrupt is consequential, the response time should be $[r \leq \frac{1}{f}]$. In some cases, missing interrupts is tolerable. For example, the WSN protocol ensures that a missed radio signal will be resent if the acknowledgment (sent from the interrupt handler) is not received. In this case, the response time can be $[r > \frac{1}{f}]$.

2.2 The Requirement for a Fault Detector

Common types of faults in WSN applications include memory access violations (e.g., buffer overflows and null-pointer dereferences), concurrency bugs (e.g., deadlocks and race conditions), and interrupt handling violations (e.g., task queue overflow and stack overflow). Based on the characteristics of these bugs, an effective static analyzer should include three important features.

Both detecting and reporting faults should be based on program paths. Given a program point where a fault occurs, not all execution paths that traverse the program point are actually faulty. In path-insensitive analysis, the information collected along different paths is merged, indicating all of the program paths contain a same property. A conservative approximation at the merge point can cause false positives, while an aggressive approximation may lead to false negatives. Path information is also important for diag-

Bug Type		Number of Bugs
Interrupt Related Bugs	Deadlock	4
	Race Condition	2
	Atomicity Violation	1
	Task Queue Overflow	2
	Stack Overflow	1
Logic Violation Bugs		5

Table 1: Interrupt-related bugs accounts for 66.7% of all severe bugs sampled from the TinyOS bug repository.

nosing and fixing the detected bug because paths explain how the fault is actually produced.

The interactions between interrupts and tasks should be modeled. In WSN applications, many of the faults, especially concurrency bugs, are related to incorrect interactions between interrupts and tasks. For example, when a global variable is accessed by multiple interrupt handlers and tasks, incorrect orders in which the variable is accessed may lead to deadlocks or race conditions. We randomly sampled 15 bugs from the most-severe-bug-pool in the TinyOs bug repository and categorized them based on their root causes. Our inspection shows that 66.7% of the sampled bugs are interrupt related (see Table 1). Among these bugs, task queue overflows occur when the processor is occupied by responding to high-frequency interrupts and no tasks can be fetched from the task queue, and stack overflows occur when interrupt handlers are continuously pushed onto the stack but not able to be executed in time.

Timing analysis should be performed. When the response time is not able to be satisfied in a WSN application, a real-time constraint violation occurs. For example, the low priority interrupts are never handled when the higher priority interrupts arrive at a very high frequency. Another example is that if it takes too long to handle an interrupt, the arrival interrupts of the same type have to be discarded. To statically determine such potential violations, we need to perform timing analysis and calculate the potential execution time for each task and interrupt handler. The worst case execution time (WCET) is often used to compare with the real-time constraints.

Figure 2 illustrates an example where timing analysis is necessary to detect a complicated bug. This bug occurred in the CC1000 radio stack and was corrected in revision 1.30 of the TinyOS-1.x source tree. In the example, `SpiByteFifo.rxMode()` at line 5 and `CC1000Control.RxMode()` at line 6 are called under case `TXSTATE_DONE` (line 3). These calls implement busy waiting, which take a long time (400 microseconds) to finish. When the post fails (e.g., the task queue is full), `RadioState` at line 9 is not reset to the idle state. Same execution paths will be taken when the application responds to next SPI interrupts at lines 13-15. Since the CPU is mostly busy waiting at lines 5 and 6 and no time to execute the task queue, the post will still fail at line 8; as a result, `RadioState`, the key state for exiting the current radio transmission, is still not set; the application enters an infinite loop. Clearly, without the knowledge of busy waiting time and SPI interrupt frequency, it is very difficult to statically detect this bug.

```
1   result_t SpiByteFifo.dataReady(uint8_t data_in) {
2     ......
3     case TXSTATE_DONE:
4     default:
5       call SpiByteFifo.rxMode();
6       call CC1000Control.RxMode();
7       bTxPending = FALSE;
8       if (post PacketSent()) {
9         RadioState = IDLE_STATE;
10    ......
11  }
12
13  TOSH_SIGNAL(SIG_SPI) {
14    ......
15    signal SpiByteFifo.dataReady(temp);
16  }
```

Figure 2: A livelock bug found in the CC1000 radio stack.

3. INTERRUPT HANDLING MODELS

To statically detect faults in WSN applications, we first need to model the control flow related to interrupt handling routines. In existing WSN applications, interrupts are handled fully preemptively in that the execution of a task or interrupt handler can be suspended at any program point, except in the atomic or interrupt-disabled sections. Verifying such a large state space is challenging. To reduce the state space a static tool handles, we present two types of lazy preemption, namely *non-preemptive* and *restricted-preemptive* interrupt scheduling models, where the interrupts are preempted only at selected program points. These program points are statically determined based on the real-time requirement. In Figure 3, we use a simple example to explain the models and discuss the tradeoffs to implement these models.

In Figure 3(a), we present control flow graphs (CFGs) for a task (left) and an interrupt handler (right). Suppose each block in CFGs is atomic, and interrupts are enabled in both routines. In fully-preemptive interrupt scheduling, every program point is preemptable, shown in Figure 3(b). We call the program points where interrupts are allowed to preempt *preemption points*. At preemption points, the control flow potentially transfers from the task on the left to the entry of the interrupt handler on the right. We use a dash line in the figure to represent the transition. In the figure, we show that if such control flow transition is considered, the program potentially executes six paths, $\langle 1, 5, (2|3), 4\rangle$, $\langle 1, (2|3), 5, 4\rangle$ and $\langle 1, (2|3), 4, 5\rangle$.

In the fully-preemptive model, the state space of a program is related to the number of types of interrupts a system supports (typically 5 to 8 types), the size of tasks and interrupt handlers, and the size of atomic sections extant in a program. We perform a study on the 29 WSN applications in the TinyOS-1.x source tree to determine the latter two sizes, using a context-sensitive and flow-insensitive static analysis. Here, context-sensitive means that we follow each function call to add up the lines of code in callees recursively; flow-insensitive means that we do not consider the exact control flow in a procedure. In Figure 4, we show that more than half of the atomic sections in the 29 WSN applications are under 20 lines of code (LOC) (with a mean of 33.5 and max of $6,197$). Figure 5 shows that the total size of tasks and interrupt handlers are much larger than the size of atomic sections (with a mean of $1,303.9$ and max of $1,563,835$). If fully-preemptive models are implemented, as in current WSN applications, the number of potential interleavings of interrupt handlers is very large.

As described in Section 1, our key observation is that in practical WSN applications, the required response time of typical interrupts is larger than the execution time of most tasks and interrupt handlers. Based on this insight, we developed the non-preemptive model, shown in Figure 3(c). In this model, the control flow transition between the task and the interrupt handler only occurs at the end of the task, indicated by the dash line between the two CFGs in the figure. At runtime, when the interrupts arrive, we cache them in the system; only when the system stack is empty, i.e., when other interrupt handlers or tasks are finished, we invoke the interrupt han-

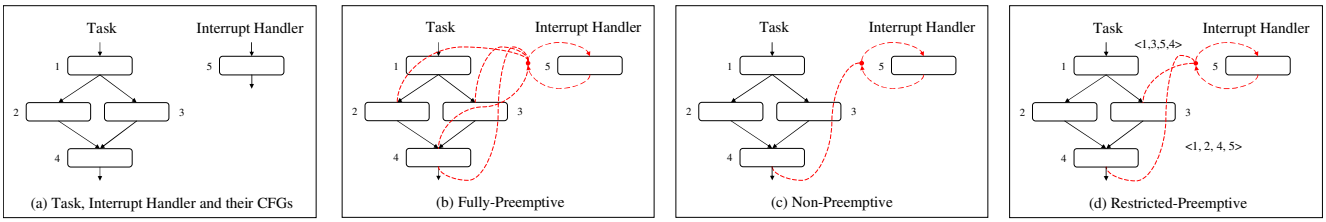

Figure 3: Interrupt handling models.

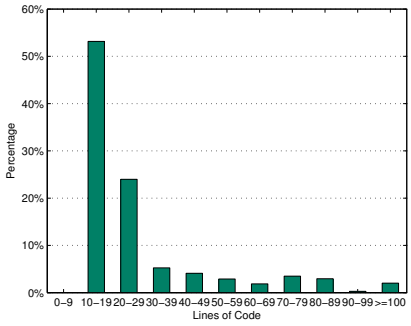

Figure 4: Size of atomic sections in terms of LOC.

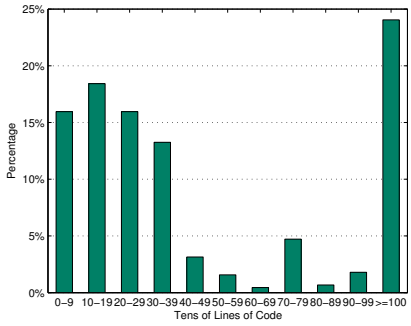

Figure 5: Size of tasks & interrupt handlers in terms of LOC.

dling routines for the cached interrupts. Adding this constraint for interrupt scheduling, we largely reduce the complexity of a program caused by the interleavings of interrupt handlers. As shown in Figure 3(c), the state space is reduced from six to two paths, $\langle 1, (2|3), 4, 5\rangle$.

In the non-preemptive model, interrupt handling is potentially delayed. The delay can cause two problems. First, the system response time decreases, which is not acceptable for critical real-time applications. Second, since the waiting time increases, the second interrupt of the same type potentially arrives before the first interrupt can be handled. As we discussed in Section 2, in WSN applications, the system is only able to record one interrupt of a type; in this case, we potentially miss interrupts, which is not always tolerable for a WSN system.

To solve the problems, we also developed the restricted-preemptive model, shown in Figure 3(d). In this model, preemption is enabled in tasks and interrupt handlers only when the real-time constraints are potentially violated otherwise. In Figure 3(d), suppose the real-time constraint cannot be satisfied along path $\langle 1, 3, 4, 5\rangle$ (e.g., block 3 takes a long time to finish) in a non-preemptive model. To resolve this real-time constraint violation, we perform a static timing analysis and determine that, to achieve the desired response time, the preemption at node 3 has to be enabled. The timing analysis reports that no preemption is required along path $\langle 1, 2, 4, 5\rangle$. We thus insert preemption points at the correspondent locations in the CFGs. The figure shows that, in the restricted-preemptive model, the state space still only contains two paths: $\langle 1, 3, 5, 4\rangle$ and $\langle 1, 2, 5, 4\rangle$.

To perform static analysis for the program that implements the models, we build an intermediate representation for the program source, namely *inter-interrupt control flow graph (IICFG)*. Examples of IICFGs are given in Figure 3. The graph consists of CFGs for tasks and interrupt handlers and also integrates a set of special edges that connect the preemption points to the entry of the interrupt handler as well as the exits of interrupt handler to the preemption points.

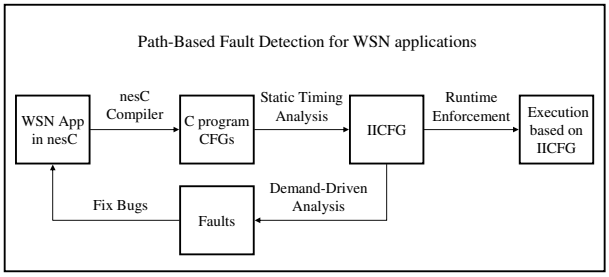

Figure 6: Work flow and components of the framework.

4. HYBRID PATH-BASED TECHNIQUES

In this section, we present a framework that applies the lazy preemption models to detect faults in WSN applications. The framework contains static timing analysis, a demand-driven, path-based bug detector, and a dynamic component that enforces the lazy preemption models for an application at runtime. In Figure 6, we show the workflow of our approach and the three main components of the framework.

Giving a WSN application written in nesC, we first compile it to a C program using the nesC compiler. CFGs of the C program are built (see Figure 3(a)). The *static timing analysis* component takes the CFGs of the program and finds preemption points in the CFGs for the correspondent types of interrupts, generating the IICFG. The *demand-driven, path-based analysis* component performs the analysis on the IICFG and detects bugs of desired types. Based on the preemption points marked on the IICFG, the *runtime enforcement* module inserts instrumentations at the specific points to enable and disable interrupts for a WSN application at runtime.

4.1 Static Timing Analysis

The goal of static timing analysis is to find preemption points in tasks and interrupt handlers, and construct an IICFG for a program. The information we use in the timing analysis includes the program source and the domain knowledge. In particular, our analysis needs

to know for each type of interrupt a WSN application supports, the arrival frequency of the interrupt, f, and the required response time, r. We also need to know the priority of interrupts. The knowledge can be found in the hardware specifications or provided by the domain experts. Since the hardware on the sensor nodes is simple and the code we see can be considered as the code we execute, we use *instruction count*, the number of instructions executed, as a metric to statically estimate the execution time for an application. Our timing analysis is conservative in that we always assume that the interrupts arrive in a worst case scenario.

In Figure 7, we use an example to explain our timing analysis technique. The example consists of a main function, a task and the handlers for interrupts A and B. Besides the source code, we also know that: 1) interrupt A arrives at frequency f_1 and requires the response time r_1; 2) interrupt B arrives at frequency f_2 and requires the response time r_2; and 3) the priority of B is higher than A.

Figure 7 gives a worst case scenario. Suppose the task starts at time s, and interrupts A and B arrive immediately after the task starts. We assume the preemption point for interrupt A is found at p. We therefore can construct the formula: $T = x + T_b + T_a$, where x is the time used to execute the task atomically between s and p, T_b is the time used to handle interrupts B_1,B_2,..., and B_m that arrive during T, and T_a is the time used to handle interrupts A_1,A_2,..., and A_n that arrive during T. Based on the formula, we build real-time constraints:

$$T = x + Tf_2h_2 + Tf_1h_1 \quad (1)$$

$$T - (Tf_1 - 1)h_1 \leq r_1 \quad (2)$$

$$T - (Tf_1 - 2)h_1 - \frac{1}{f_1} \leq r_1 \quad (3)$$

$$\vdots \quad (4)$$

$$T - \frac{Tf_1 - 1}{f_1} \leq r_1 \quad (5)$$

$$T - Tf_1h_1 - (Tf_2 - 1)h_2 \leq r_2 \quad (6)$$

$$T - Tf_1h_1 - (Tf_2 - 2)h_2 - \frac{1}{f_2} \leq r_2 \quad (7)$$

$$\vdots \quad (8)$$

$$T - Tf_1h_1 - \frac{Tf_2 - 1}{f_2} \leq r_2 \quad (9)$$

Here, equation (1) is derived by replacing T_b with Tf_2h_2 and T_a with Tf_1h_1 in the formula $T = x + T_b + T_a$. Tf_2 and Tf_1 are the numbers of interrupts that can arrive during T for interrupts B and A respectively. h_2 and h_1 are the time needed to handle the two interrupts, which can be statically computed based on the program source. Constraints (2) to (5) specify that for each interrupt of type A, the actual response time, calculated using the time when the handler is completed minus the time when the interrupt arrives (see the left of the inequality), should be less than or equal to the required response time r_1. Similarly, constraints (6)-(9) are real-time requirements for handling interrupts of type B.

Given the set of constraints shown above, our goal is to compute the maximum value of x. If such an x never exists, the preemption points cannot be found, the program contains a bug and the real-time constraints will be violated at runtime. If x is larger than or equal to the WCET of the task, the non-preemptive model can be applied, and the preemption point can be added at the end of the task; $Tf_2 + Tf_1$ is the maximum storage overhead for caching the interrupts. If x is otherwise less than the WCET, preemption points should be inserted along paths in the task. We perform instruction counting along all execution paths for the task and determine the

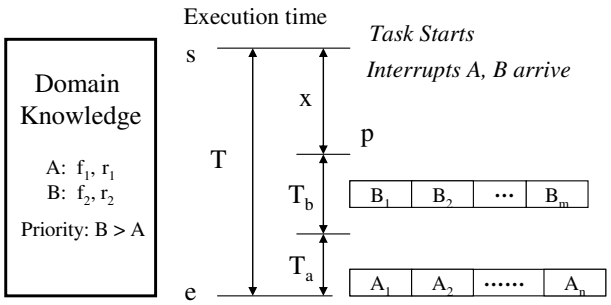

Figure 7: Static timing analysis.

preemption points based on x. Loops and the special instructions such as *sleep* are modeled to get conservative instruction counts. One approach is to use a constant or upper bound for loop iterations. Based on the detected preemption points, we then construct the IICFG.

4.2 Runtime Enforcement

Our dynamic component enforces the interrupt scheduling based on the computed preemption points. It accomplishes the following three tasks.

First, we split all interrupt handlers in an application into two parts, namely *record handler* and *action handler*. The record handler caches an incoming interrupt, and more specifically, it records the data collected from any hardware ports regarding the interrupt. It is invoked whenever an interrupt arrives. The action handler is called when the interrupt is enabled at the preemption points. It reads the cached data and accomplishes the actions implemented in the original handler.

Second, at each preemption point, we use the classic injection-based instrumentation technique to fire cached interrupts. We replace the original instruction at the preemption point with a jump to an extra piece of code we inserted into the binary, referred as a *trampoline*. The trampoline checks the cached interrupts and invokes the action handlers accordingly. After that, it executes the original instruction, and jumps to the instruction that immediately after the preemption point.

Third, when the program is initially loaded onto the sensor nodes, values in the interrupt vectors are modified to point to the corresponding record handlers instead of the original handlers.

The net effect of the above process is that when a program executes, interrupts can happen at any time and the input data is recorded. At any preemption point, the recorded interrupts are eventually fired through action handlers.

It should be noted that the above techniques are applicable with the assumption that the interrupts arrive in a fixed frequency. If the interrupts are triggered by aperiodic events, we take one of the following two approaches: 1) for systems where violations for real-time constraints are not critical, we will compute preemption points using an estimated frequency (e.g., 500 Hz) and response time (e.g., 1 msec); and 2) for timing critical systems, we will invoke action handlers at a constant frequency, rather than at the preemption points (computed via estimated frequency and response time), to response the cached interrupts. The more frequently the handler is invoked, the more runtime overhead it incurs and the less likely the real-time constraints will be violated.

4.3 Demand-Driven Analysis to Detect Faults

Since our dynamic component enforces a restricted interrupt handling schedule, some of the concurrency bugs caused by interleav-

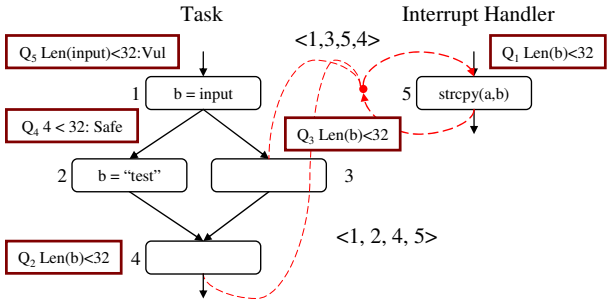

Figure 8: Detecting buffer overflows on the IICFG.

ings that are not allowed in the new scheduling are prevented. In this section, we present a demand-driven, path-based analysis that detects memory access violations, interrupt handling violations, and concurrency bugs that are not able to be prevented from the new interrupt scheduling.

Take buffer overflow detection as an example. Our demand-driven analysis first scans the IICFG of a program and identifies buffer access statements. A query is then raised, inquiring whether a buffer overflow can occur at the statement. A backward analysis is performed from the statement, along program paths towards the entry of the main function. The information that is useful to determine the resolutions of the query is collected. The analysis terminates when the query is resolved.

In Figure 8, we show how the path of a buffer overflow is discovered by our demand-driven analysis. In the first step, our analysis finds a buffer write statement, *strcpy(a,b)*, in the interrupt handler. Suppose here, variable *a* is a local buffer with the size of 32 bytes, while variable *b* is a global buffer which can be modified outside the interrupt handler. To determine if the buffer access is safe, we construct query $[len(b) < size(a)]$ at node 5. In this case, the size of buffer *a* can be determined statically, and we therefore update the query to $[len(b) < 32]$. The query is propagated backwards, first through edges $\langle 5, 3\rangle$ and $\langle 5, 4\rangle$, and arrives at node 3 and node 4 in the task. At node 3, the query is advanced to node 1; the analysis finds that buffer *b* is assigned by the input package without bounds-checking. A buffer overflow is found. Path $\langle 1, 3, 5\rangle$ is reported as faulty. Along branch $\langle 4, 2\rangle$, we collect information at node 2 and determine path $\langle 2, 4, 5\rangle$ is safe.

5. RELATED WORK

Static techniques for WSN applications mainly include model checking and dataflow analysis. A category of model checking techniques focus on verifying the distributed algorithms of WSNs, rather than the correctness of software running on individual sensor nodes. In order to reduce the state space, these techniques usually abstract away the details of single device, such as interrupts [1, 4, 8, 9]. Our research is orthogonal in that we develop techniques to analyze software on sensor nodes.

Analyses for software running on single sensor nodes have two approaches. One approach is to apply traditional dataflow analysis, which is path-insensitive and thus imprecise [2, 3, 11]. Another approach is to randomly select a limited number of paths and conduct precise analysis on them [12]. According to the best of our knowledge, our work is the first that performs path-based analysis at whole program scale for WSN applications, and we reduce state space by applying a timing analysis and a demand-driven algorithm.

Techniques for applying restricted preemption scheduling have been used in testing [10]. The goal of this work is to ensure that randomly generated scheduling is valid, and their approaches are purely dynamic. Lai et al. represent interrupt preemptions on the control flow graph by adding preemption edges; however, no timing analysis is performed to statically schedule the interrupt handling [5].

6. CONCLUSIONS AND FUTURE WORK

Program paths are important for precisely detecting faults and also presenting useful information for diagnosing bugs. However, statically identifying bugs from WSN applications based on program paths is hard due to a potentially exponential state space caused by non-deterministic interleavings between sequential executions and various interrupts. Based on how preemption is enabled in tasks and interrupt handlers, we develop lazy preemption models, which can greatly reduce the state space needed for analysis. We apply timing analysis to determine interrupt scheduling based on these models, and develop a lightweight runtime monitor to enforce the scheduling dynamically. We designed a demand-driven, path-based static analysis to detect faults on the programs implemented with the models. In the future, we plan to model a set of faults presented in Section 2.2, and perform bug detection experiments on the framework. We aim to collect the bug detection rates as well as both static and dynamic overhead of our approach.

7. REFERENCES

[1] P. Ballarini and A. Miller. Model Checking Medium Access Control for Sensor Networks. In *ISoLA '06*, 2006.

[2] D. Brylow, N. Damgaard, and J. Palsberg. Static Checking of Interrupt-Driven Software. In *ICSE '01*, 2001.

[3] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient Memory Safety for TinyOS. In *SenSys '07*, 2007.

[4] C. Killian, J. Anderson, R. Jhala, and A. Vahdat. Life, Death, and the Critical Transition: Finding Liveness Bugs in Systems Code. In *NSDI '07*, 2007.

[5] Z. Lai, S. Cheung, and W. Chan. Inter-Context Control-Flow and Data-Flow Test Adequacy Criteria for nesC Applications. In *FSE '08*, 2008.

[6] W. Le and M. L. Soffa. Marple: A Demand-Driven Path-Sensitive Buffer Overflow Detector. In *FSE '08*, 2008.

[7] W. Le and M. L. Soffa. Path-Based Fault Correlation. In *FSE '10*, 2010.

[8] P. Li and J. Regehr. T-Check: Bug Finding for Sensor Networks. In *IPSN '10*, 2010.

[9] L. Mottola, T. Voigt, F. Osterlind, J. Eriksson, L. Baresi, and C. Ghezzi. Anquiro: Enabling Efficient Static Verification of Sensor Network Software. In *SESENA '10*, 2010.

[10] J. Regehr. Random Testing of Interrupt-Driven Software. In *EMSOFT '05*, 2005.

[11] J. Regehr, A. Reid, and K. Webb. Eliminating Stack Overflow by Abstract Interpretation. *ACM Trans. Embed. Comput. Syst.*, 4(4), 2005.

[12] B. Schlich. Model Checking of Software for Microcontrollers. *ACM Trans. Embed. Comput. Syst.*, 9(4), 2010.

[13] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: A Comprehensive Source-Level Debugger for Wireless Sensor Networks. In *SenSys '07*, 2007.