2015 IEEE 21st International Conference on Parallel and Distributed Systems

Stethoscope: A Sustainable Runtime Debugger for Wireless Sensor Networks

Yuan He
School of Software and TNLIST, Tsinghua University
Beijing, China
he@greenorbs.com

Shuo Lian
WuXi Tsinghua IOT Center
Wuxi, China
alian2015@qq.com

***Abstract*—Debugging wireless sensor networks (WSNs) is notoriously difficult, due to the resource constraints on the sensors and distributed running of the debugged programs. Many bugs only manifest themselves during the actual operation of a network, thus requiring runtime debugging of the sensor program. A WSN debugger has to meet two important design criteria, namely saving energy and preserving responsiveness to normal system/network events during debugging. In this paper, we propose Stethoscope, a sustainable runtime debugger for WSNs. In Stethoscope, we devise a new technique called Quick Switch, which enables dynamic binary instrumentation in the RAM instead of the program flash. By incorporating a light-weight hooking mechanism, Stethoscope ensures runtime responsiveness of the debugged program. We implement Stethoscope and demonstrate its advantages with respect to efficacy, energy consumption, and memory cost.**

***Keywords*-Debug; Sensor Network; RAM; Energy;**

I. Introduction

Software debugging is crucial for ensuring the correctness and efficacy of computer programs. As a type of network that closely interacts with the physical environment, wireless sensor networks (WSNs) update their programs at a relatively high frequency, due to reasons like bug fixing and functionality upgrade. Debugging thus becomes a crucial task that needs to be conducted throughout the operational lifespan of a WSN.

Due to the distributed nature of WSNs [1], software debugging becomes a complex and challenging issue. The common-off-the-shelf sensor motes are usually constrained with respect to power source and memory space. A sustainable WSN debugger actually means the extremely low cost of debugging with respect to energy and storage. The frequent debugging must not add up too much power consumption or memory cost. Another challenge is that debugging must be conducted during the runtime of a WSN, i.e. when the WSN is deployed and operates in realistic context. In such a networking system that inherently interacts with the physical world, many program bugs only manifest themselves during the runtime and cannot be uncovered by offline tests. Then it incurs another extremely difficult problem, namely how to make a debugged program correctly responsive to both the debugging commands and normal system/network events.

Some existing approaches for WSN debugging propose to wire additional debugging device [2] to the debugged sensor nodewhich are clearly labor-intensive and cannot scale with the network size. Another class of approaches debug in an offline manner [3]. Under pre-configured debugging contexts, those approaches debug programs with symbolic inputs. Such approaches are likely to miss many critical bugs that only appear during the network runtime. Recent works propose to insert debugging agents into the source code [4]. By inputting debugging commands at the sink side, a programmer may control the debugging process via the agents, who are not easily adapted to online modifications of debugging tasks. Moreover, the agents are likely to intervene into the normal running of the debugged programs, potentially leading to Heisenbug problems [5].

In this paper we propose Stethoscope, a sustainable runtime debugger for WSNs. Stethoscope incorporates two novel techniques, Quick Switch and Hooking. Quick Switch makes dynamic binary instrumentation in the RAM space and enables fast transitions between contexts of debugging and normal program execution. The light-weight Hooking mechanism reinforces the debugger with a customizable interface to define program behavior under debugging commands and normal network/system events. Our contributions can be summarized as follows.

1. We propose Quick Switch, a novel technique to perform dynamic binary instrumentation in the RAM space. This makes the debugged program component(s) completely run in the RAM, saving a lot of energy from flash writes/erases.

2. We devise the Hooking mechanism to include redefined interrupt processing functions in the debugger. The debugger is thus able to respond appropriately to normal system/network events even when a debugging command is being executed.

3. We implement Stethoscope on common-off-the-shelf sensor motes and conduct extensive experiments to demonstrate the efficacy and efficiency.

The rest of this paper is organized as follows. In Section II we discuss the related work. Section III elaborates on the design of Stethoscope. Section IV presents the evaluation results. We conclude in Section V.

II. Related Work

The existing WSN debugging approaches can be classified into three categories.

The first category mainly does offline debugging, namely testing the functionalities of programs before the sensor node is practically deployed. A typical example is KleeNet [6], which is a debug environment running unmodified sensor network programs on symbolic inputs. It can automatically injects non-deterministic failures to simulate network runtime conditions. Other related works include FSMGen [7], T-check [8] and so on. Those approaches have limited efficacy for ensuring program correctness in practice, because they are unable to debug the networking behavior of sensor programs. They also fail to consider various uncertain factors in real deployments, e.g. environmental dynamics.

The second category is remote debugging [4], [9], [10], [11], which is indeed desired by WSN applications. Dust-Miner [12] and LiveNet [13] eavesdrop on messages in the network for visibility into network operations and interactivities without consuming much node resources. EnviroLog [14] logs non-deterministic events to produce efficient in-network execution replay. Sympathy [15] collects a small amount of data to identify the cause of network failures. Marionette [16] provides remote access to source-level symbols and statements in the source code of programs on the sensor nodes. Those approaches all require certain amount of prior knowledge, especially concerning the potential bugs and anomalies, by nature limiting their ability to uncover unknown bugs.

The third category is based on the technique of binary instrumentation. Typical examples include Clairvoyant [4] and Declarative Tracepoints [17]. They both include an agent-like module on the remote sensors to receive and interpret debugging commands from the sink. Declarative Tracepoints provides a programmable interface for describing debugging operations. The associated information can be downloaded and executed on the nodes at runtime, enabling convenient remote debugging. Clairvoyant [4] generates a modified binary, where the debugging modules are inserted. The binary are then burned to the program flash on sensors before deployment.

There are some other related works, which do not belong to debugging approaches but also have certain effect in improving the program correctness. Using compiler [18] and virtual techniques, a debugger can debug a wired sensor node connected via the serial port and virtualize a network for debugging at the same time. Safe TinyOS [19] is a compiler toolchain that executes mandatory memory check for variables and modules defined in user programs.It may warn a developer about unsafe code and instruments the code with safety annotations. The work in [20] proposes a discrete event simulator to support functional debugging for TinyOS-based programs, which allows a user to compile the nesC source code and run the program under TOSSIM. Those approaches can simulate the runtime of a sensor node to a certain extent [21], but fail to characterize the runtime interactivities among multiple sensor nodes in the network [1]. As a result, programs passing tests are not guaranteed to behave correctly.

III. Design

We start with a brief overview of the design architecture and work flow of Stethoscope. Then we respectively introduce the two core techniques, namely Quick Switch and Hooking. Some implementation details like how a debugging command is processed are introduced in Section III-D. Other important issues related to the design will be discussed in III-E.

A. Overview

Stethoscope mainly consists of three components: binary generator, command generator, and debug agent.

The binary generator is on the programming host, where the binary program image is generated and deployed to the sensors. The command generator is on the sink. It generates debugging commands at runtime according to the programmers debugging purposes. In general practice, the host and the sink are usually a same computer that is connected to the rest of a WSN. The program binary images may be either burned to the sensors before deployment or delivered to the sensors via wireless reprogramming[22].

The debug agent is on every sensor, which is used to receive and interpret the debugging commands and perform corresponding operations. The modules of Quick Switch and Hooking are included in an agent.

For clarity in introducing the design and implementation of Stethoscope, we use TinyOS 2.1.x and TI MSP430 16xx series MCU as the example software and hardware platforms. Later in the subsection of discussion, we will show that the design and implementation can be easily generalized to many different sensor platforms.

At the host side, the binary generator modifies the original binary program image, changing direct function calls into indirect ones and adding a switch table to support debugging context switching. Sensors programmed with the modified binary image are deployed for normal operation. When the programmer wants to debug the program, the command generator generates a debugging command according to the programmers input and sends it to the target nodes. On receiving the command, the debug agent residing in a target node accordingly performs debugging operations. Multiple sensor nodes in the network can be debugged simultaneously.

At the sensor side, the debug agent plays the role of managing the debugging process. It has an interpreter, which parses the received debugging commands and calls different executors to perform corresponding operations for different commands. For example, upon receiving a watch, log, or other similar commands, Stethoscope accordingly modifies the code in RAM (socalled dynamic binary instrumentation in the RAM), which is much easier and more light-weight than modifications to the program flash. For the watch command, Stethoscope finds the address of the variable to

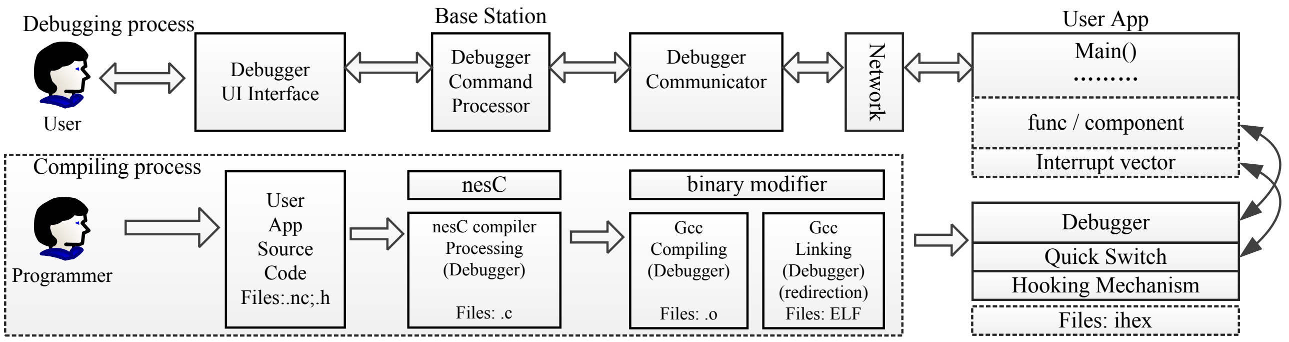

Figure 1. Illustrates the workflow of Stethoscope.

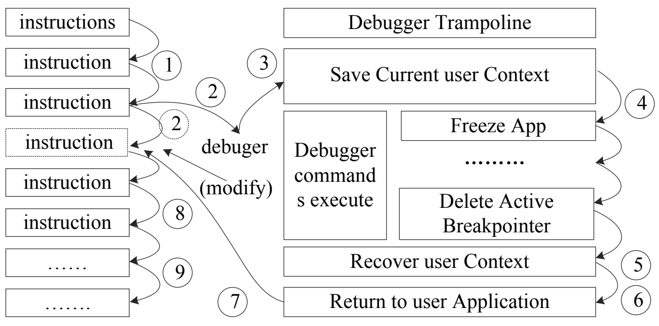

Figure 3. Conventional DBI for debugging.

watch and transmits the value back to the base station. When a step or continue command is received, the control needs to be transferred back to the user application. In that case, a traditional debugger modifies the program flash by replacing the jmp instruction with the original instructions. In Stethoscope, the control is transferred back to the user application by directly restoring the address in the switch table. No writes/erases on program flash are required.

B. Quick Switch

In order to illuminate the advantages of Quick Switch and the idea behind it, we first introduce the design of conventional DBI (Dynamic Binary Instrumentation) in Figure 3. DBI is a method of analyzing the behavior of a binary program at runtime through the injection of instrumentation code. The injected instrumentation code executes as part of the normal program. Using DBI, a programmer can dynamically insert debugging operations into the binary program image, however, at the cost of frequent write/erase operations to the program flash. When a $breakpoint$ command is received, control is transferred to the debugger. To insert the breakpoint, a conventional DBI approach replaces the instruction at the breakpoint line with a jmp instruction, which points to a $trampoline$ in the program image. When the breakpoint is reached at runtime, the program jumps to the trampoline, which then (i) saves application context, (ii) executes instrumented code for debugging, and (iii) jumps back to the application code after command finishes. Such debugging techniques require writing program flash multiple times for a single debugging command, which incurs excessive time/power overhead.

Quick Switch changes the above workflow of DBI and performs debugging operations without any program flash modification. That means potential energy saving by orders of magnitude. To achieve this goal, Quick Switch generates in the RAM a mirror image of the debugged code block and modifies the mirror image according to debugging requirements. Now that the debugged code block is kept in the RAM, we need to address an immediate question: how to make the runtime of the debugged program switch between flash and RAM, without affecting the continuous program execution?

To solve this question, Quick Switch incorporates a two-step approach.

First, let's trace back to the source code compilation phase. Quick Switch generates a switch table according to the code. It analyzes the code, identifies all the function calls to the functions that need to be debugged, changes those direct function calls into indirect function calls. The true function addresses are stored in the switch table while the indirect function calls are made pointed to the addressed in the switch table.

Second, when a debugging command is received during the program runtime, Quick Switch changes the function address in the switch table, making it point to the new function address in the RAM. By doing this, any call to this function during the debugging process will be quickly directed to the new modified function. Accordingly, program runtime is switched from the program flash to the RAM. When the debugging process finishes, the address in the switch table is restored, making a function call quickly switched back to the original function in the program flash.

For better understanding of the Quick Switch technique, Figure 2 plots the debugging process with Stethoscope. When a breakpoint is set, instead of replacing the instructions in the program flash, Stethoscope (i) locates the function which contains the breakpoint, (ii) saves the corresponding function address in the switch table, (iii) makes a copy of the function in the RAM and accordingly modifies it and (iv) changes the corresponding function address in

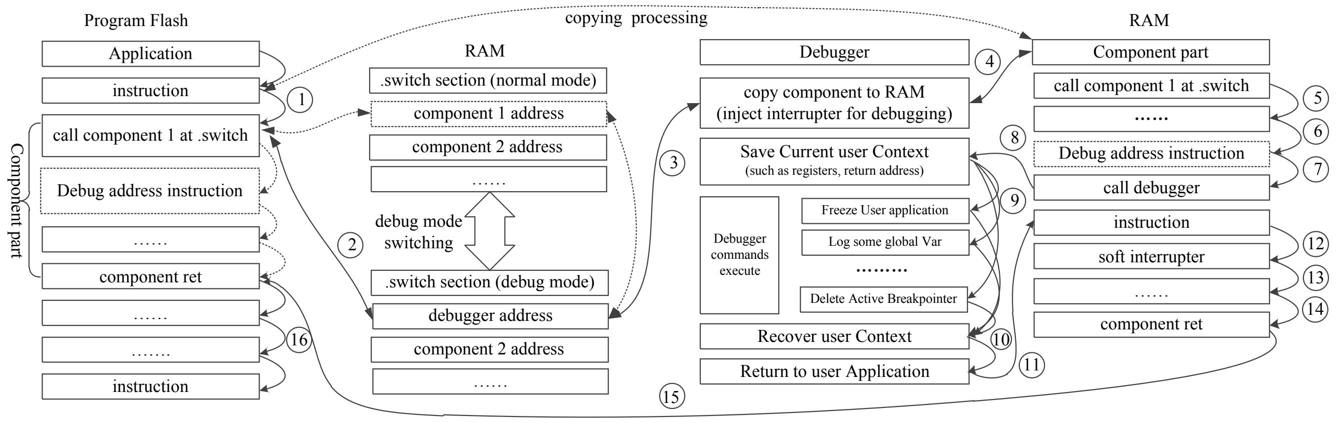

Figure 2. Quick Switch for debugging with Stethoscope.

the switch table, directing it to the new function location in the RAM. When the breakpoint is reached during program runtime, the modified function in the RAM will be executed.

From the above description, we can see Quick Switch realizes continuous program execution that dynamically switches between the program flash and the RAM. Because program execution in the RAM is generally very fast, Quick Switch incurs extremely low processing overhead. More importantly, the debugged code, as well as the debugging operations on it, is executed entirely in the RAM, saving a large amount of energy from program flash writes/erases.

C. Hooking

A common challenge of debugging tools is to deal with interrupts. Generally, when the program runs in the debug mode, a single interrupt with higher priority may block the current operation in the debugging process and lead to unexpected consequence.

For example, when the program runs in the debug mode, a hard interrupt preempts the current debugging process, changes the program state, and results in unexpected execution path as shown in Figure 4. It is worth noticing that simply disabling interrupts cannot solve the problem, because interrupts of debugging components will be disabled as well. The debugger also needs to receive debugging commands. Disabling radio interrupts makes the node unable to behave correctly upon receiving debug commands. Actually, disabling interrupts is likely to isolate the debugged sensor node from the rest of the network, failing to meet the runtime debugging conditions.

In order to solve the above problem, we propose the Hooking mechanism. It analyzes the executable program after the compilation process. Then it changes the interrupt vectors, making them point to interrupt processing functions in the debugger module of Stethoscope. Hooking actually "hijacks" the interrupt handling process as a step before the interrupt handler is actually triggered.

To avoid loss of any incoming interrupt during debugging, we have analyzed all kinds of interrupts and invoked

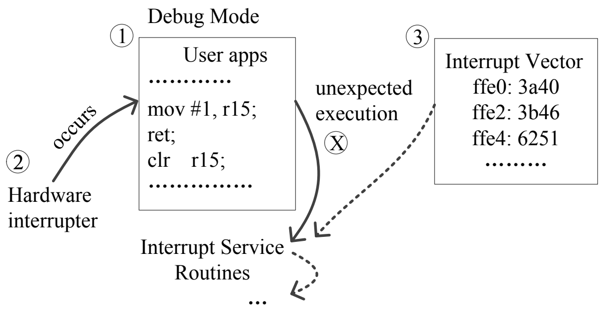

Figure 4. A hardware event invokes an unexpected execution.

corresponding processing functions in Stethoscope. In this way, Stethoscope is able to control the processing flow of all kinds of interrupts and direct them to the right handlers. Figure 5 shows the detailed workflow of Hooking. When a hard interrupt event occurs, the modified interrupt vector may lead the program to the handler which can handle this interrupt event correctly. As shown in Figure 5, only a redirection module and a Hooking module need to be added, which count for 276 bytes of memory. The additional processing delay of Hooking is nearly negligible, according to our experimental results presented later.

D. Implementation Issues

Switch Table: Here we introduce the details of generating a switch table. First we introduce the memory allocation of a typical program. Normally, a program contains the .text section (for program instructions), .data section (for initial values of data in the program) and .vector section (containing the addresses for interrupt handlers). New sections may be added by special compilation options.

Stethoscope first analyzes the function table generated from the code. Based on the function table, Stethoscope knows the corresponding address of each function. Then, Stethoscope processes the program (.text) section, finds all the direct function calls and substitutes them with indirect

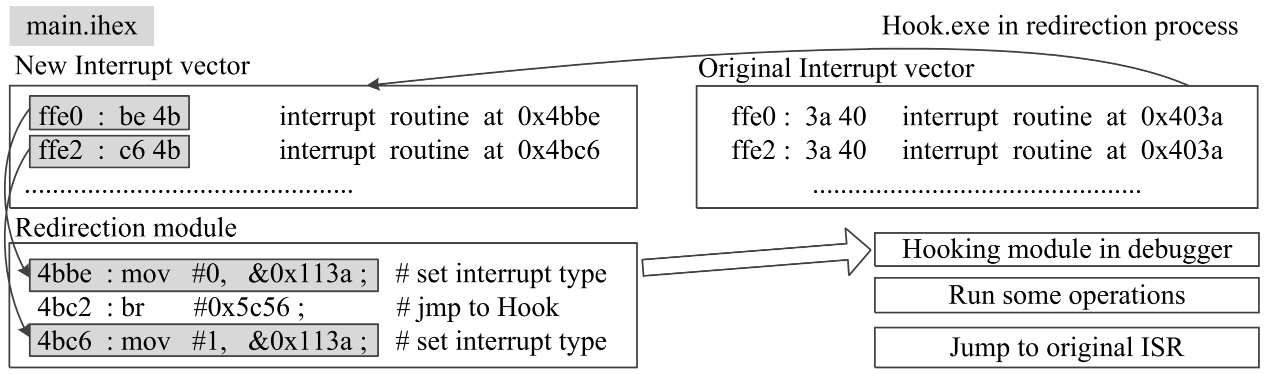
main.ihex
Hook.exe in redirection process
New Interrupt vector
ffe0 : be 4b interrupt routine at 0x4bbe
ffe2 : c6 4b interrupt routine at 0x4bc6
Original Interrupt vector
ffe0 : 3a 40 interrupt routine at 0x403a
ffe2 : 3a 40 interrupt routine at 0x403a
Redirection module
4bbe : mov #0, &0x113a ; # set interrupt type
4bc2 : br #0x5c56 ; # jmp to Hook
4bc6 : mov #1, &0x113a ; # set interrupt type
Hooking module in debugger
Run some operations
Jump to original ISR

Figure 5. Hooking Mechanism for debugging with Stethoscope.

ones. Stethoscope places the original address of a function in the switch table and makes the indirect function call point to the corresponding address in the switch table. In this way, the indirect function call can invoke the right function, while no program flash write is required. We use the binary program image generated after this step to program a sensor node.

The switch table is stored in a new section (.switch) which can be defined in the linking process. The mspgcc linker gives the user convenient interfaces to modify the definition of sections in the program image. The address range of (.switch) section can be easily calculated according to the end address of .data section, and the size of the switch table.

Interrupt Hooking: Based on the hooking mechanism, Stethoscope modifies the original interrupt vectors with a set of meticulous values which record the interrupt type and direct to the Stethoscope interrupter handler. The detail is as follows: Stethoscope uses a tool ($hook.exe$) to modify the original interrupt vectors with a redirection module. In the redirection module, Stethoscope first records the interrupt type, and then jumps to the hooking module which runs some operations if demanded, and returns to the original interrupt service routine.

For example in Figure 5 , in the first step, the original interrupt vector value is modified from 0x403a to 0x4bbe which directs to the redirection module. Then the redirection module saves the interrupt type in 0x113a and jumps to the hooking module at (0x5c56). In this way, every interrupt event is handled by Stethoscope before the original interrupt service routine starts, which guarantees the correctness of debugging process as well as the responsiveness to normal system/network events.

E. Command Generator

A debugging command usually specifies a location in the source code and the corresponding debugging operations. For example, to set a breakpoint at a line, the programmer (the person who debugs the program) needs to decide the line number in the source code. According to the line number, the command interpreter at the host side determines the function where the specified code line lies (we call it *host function* of the code line). Then the name and address of the host function is determined, which is stored in the switch table. After that, the line number is translated into an *offset* to the starting address of the host component, and the $setbreakpoint$ command is translated into a two-element tuple <*host_function_ID, offset*>. When a debugging command is received, Stethoscope finds the function address from the switch table according to *host_function_ID*. *offset* is used to calculate the concrete location of the instructions to break.

F. Discussion

1) Extra Advantage of RAM-based Debugging: The voltage requirement of the existing WSN debugging tools is often too high, because they mostly require frequent writes to the program flash, while the latter generally has to work under a higher voltage. Consequently, during the major portion of the batterys lifespan, the debugging tool simply cannot work. Stethoscope enables debugging in the RAM, typical lowering down the required voltage from 2.4v to 1.8v.

2) Generalizability: There are some sensors that do not support program execution in the RAM, e.g. the early MiCA series using Atmega128L microcontroller [23]. Those sensors are not widely applied nowadays and not deemed as the mainstream of future sensor platforms.

As for the supply voltage issue of RAM-based debugging, we find through survey that many currently applied sensor platforms have a lower voltage requirement to execute programs in RAM than to programming the flash memory. (e.g. MSP430AFE2x series, MSP430G2x series) [24]. However, the benefit of Stethoscope in saving energy/time consumption of program flash writes/erases is still remarkable, as we will demonstrate in the evaluation section.

IV. Evaluation

In this section, we evaluate the performance of Stethoscope in terms of different metrics, such as memory cost, computational overhead, energy consumption, and lifespan. We also compare Stethoscope with the DBI based approach,

Table I
Memory cost of different components of Stethoscope. (BlinktoRadio [25])

Telosb Mote	BlinktoRadio	Project_1
# of the other components in the source code	5	10
Switch table (RAM)	10 Bytes	20 Bytes
Hook mechanism	276 Bytes program flash	276 Bytes program flash
Stethoscope main module	~10 KB program flash	

Table II
Comparison in switch overhead between Stethoscope and a DBI-based approach.

Telosb Mote	DBI-based	Stethoscope
Save registers	~0.0177 ms	~0.0177 ms
Recover registers	~0.0156 ms	~0.0156 ms
Switch interrupt vector	~3.48 ms	~$3.05 * 10^{-4}$ ms (use Hook)
Switch to debug mode	~35.76 ms (use DBI)	~0.018 ms (use QS)
Switch to normal mode	~19.34 ms (use DBI)	~0.016 ms (use QS)

Table III
Comparisons in the times of erase/write program flash between DBI and Stethoscope, and the times of RAM operations increased by Quick Switch.

Commands	Erase/Write Program Flash **DBI-based**	Increased RAM operations by Quick Switch **Stethoscope**
connect *id*	0	0
disconnect *id*	0	0
continue	(IP, IP)	<=ND
step	(1MI, 1MI)	0
stepi	(1, 1)	0
next	(UI, UI)	1
breakpoint *[file:]line|func*	(IP, IP)	<=ND
condition *[n] [expr]*	(IP, IP)	<=ND
watch *expr*	(2MI, 2MI)	0
delete *[n]*	(IP, IP)	<=ND
print */format [expr]*	0	0
set *var=expr*	1	0
call *func*	(IP, IP)	0
frame *[n]*	0	0
list	0	0
log	(IP, IP)	<=ND
radio	(IP, IP)	<=ND

IP: The number of active insertion points; **MI**: The number of machine instructions executed; **UI**: The number of statically unpredictable control-transfer instructions; **ND**: The number of the components in source code.

the technique used in state-of-the-arts debugging tools. The sensor node we use in the experiments is TelosB mote with MSP430F1611 MCU, 10KB RAM and 48KB program flash.

A. Memory Cost

We first measure Stethoscope's memory cost in both the program flash and the RAM. In our implementation of Stethoscope, the debugger's main module on the sensor node only occupies about 10KB program flashes, as shown in Table I. In comparison, a GDB-based debugger typically needs at least 33KB of program flash memory. As for the RAM, we compare the memory cost when debugging different programs with Stethoscope. Specifically, the switch table of Stethoscope incurs very low RAM cost (each function requires only 2 Bytes of RAM). We can see from Table I that debugging programs of BlinkToRadio [25] and Project_1 (the program used in our previous measurement study [26]) only cost 10 Bytes and 20 Bytes of RAM space, respectively.

We also measure the memory cost of the hooking module. Still using BlinkToRadio and Project_1 programs as examples, we can see that the hooking module only costs 276 Bytes of RAM in both scenarios. This is because the MSP430F1611 only has 16 registers, resulting in a constant memory cost in RAM.

B. Computational Overhead

The computational overhead of Stethoscope mainly consists of two parts: One of them is the overhead incurred by switching between different modes and contexts for debugging purposes. The other is the overhead of executing debugging commands.

We first evaluate the switching overhead, including the overhead to save registers, recover registers, switch interrupter vector, switch to the debug mode, and to switch to the normal mode. The comparison results between the DBI-based approach and Stethoscope are shown in Table II. The overhead is quantified in terms of the processing latency.

We may notice that the two approaches have the same processing overhead in register-related operations, because those two factors are determined by the capacity of MCU. Another important result is that Stethoscope significantly saves the processing overhead in switching. Specifically, the total switching overhead of the DBI-based approach is around 58ms. In comparison, the switching overhead of Stethoscope is less than 0.035ms. The great advantage with respect to switching overhead is mainly because Stethoscope does not need to write program flash for switching, but the DBI-based approach needs. Processing on RAM is naturally much faster and more efficient than processing on the program flash.

We continue to compare the overhead to execute debugging commands, measured in the number of program flash writes and the number of RAM operations. The comparison results in Table III show Stethoscope does not need to modify the program flash at all.

Nevertheless, quick switch in Stethoscope does induce some operations on the RAM. For comprehensive understanding of the computational overhead, we list the number of RAM operations incurred by quick switch in Table III. We can see quick switch incurs very few RAM operations. In total, the number of RAM operations incurred by any debugging command under quick switch is not more than the total number of components in the source code, because such an amount of RAM operations are sufficient to shift all components into RAM. Moreover, the RAM operations are by nature much more efficient and energy-saving than operations on the program flash.

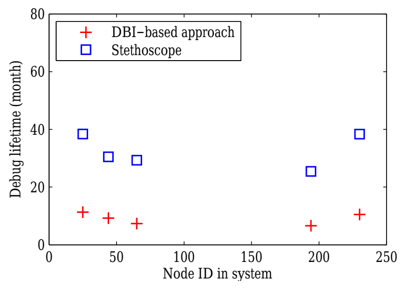

Figure 8. Comparison of debugger life time in a real system.

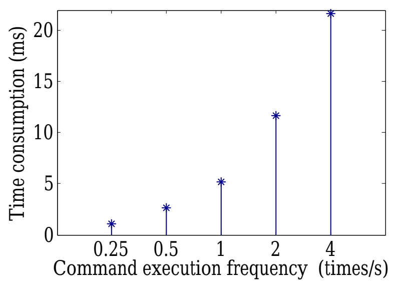

Figure 9. Execution cost of command *radio* in different frequency.

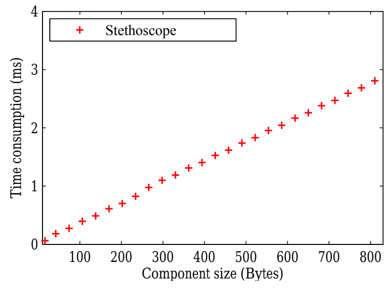

Figure 10. Execution cost of injecting a breakpoint in a debugged component of different sizes.

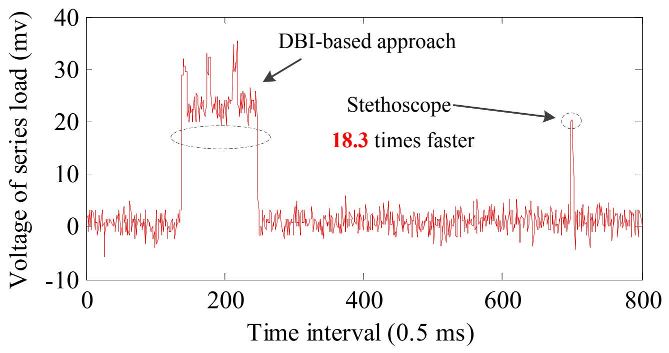

Figure 6. Instantaneous voltage of debugger operation with Stethoscope and the DBI-based debugger captured by an oscilloscope.

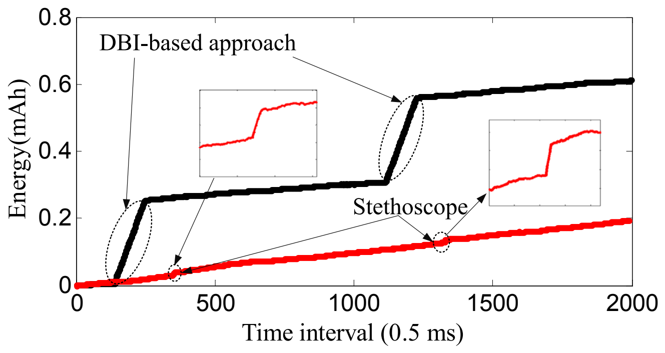

Figure 7. Accumulative energy consumption of essential debugger operation with Stethoscope and the DBI-based approach.

C. Energy Consumption

Now we evaluate the energy consumption of Stethoscope for executing a debugging task, which mainly consists of four steps: context switch, debug mode switch, execution of the debugged component (about 1000 instructions in the test case), and user mode switch. In order to make a fair comparison between Stethoscope and the DBI-based approach, we use a typical testing series circuit to compare the instantaneous voltage along the same task execution process.

Figure 6 shows that it takes 2.99 ms for Stethoscope to finish the above test case, while it takes 54.7 ms for the DBI-based approach to finish the same task. Stethoscope is **18.3** times faster than the DBI-based approach. Compared with the DBI-based approach (0.2483mAh), Stethoscope reduces the energy consumption by **96.69%**, consuming only 0.0082mAh, as shown in Figure 7. In other words, Stethoscope not only executes the debugging tasks very fast, but also significantly saves energy consumption.

D. Lifetime of Debugger

According to our previous discussion, on many sensor platforms Stethoscope can extend the debugger lifetime, because it has relatively low voltage requirement. Figure 8 presents an intuitive evaluation result of the debugger lifetime on several randomly selected nodes in Project_1. The sensor node is TelosB mote with two AA batteries. As we can see from the figure, Stethoscope extends the debugger lifetime for at least three times, compared with the debugger lifetime using DBI-based technique.

E. Impacting Factors

In this section, we evaluate the impact of two factors when applying Stethoscope to real WSNs. The two factors are the execution frequency of debugging commands and the size of the debugged component.

Figure 9 shows the relation between the time consumption and the execution frequency of debugging commands. Using the radio command as an example, we can see that even performing radio operations 4 times per second, the total time to finish executing the 4 operations (22.66 ms) is still acceptable. The average time consumption of one time execution of a command is around 5ms. That meets the requirement of a wide variety of scenarios and can be applied to computation intensive applications.

F. On the Size of Debugged Component

Figure 10 shows the relationship between the component size and time consumption for injecting a breakpoint. Intuitively, the time consumption will increase when the size of the debugged component increases, because a larger component needs more copy operations on the RAM. But it is worth noticing that operations on the RAM are extremely fast and the time consumption to inject a breakpoint is only a few milliseconds. The result in Figure 10 indicates that the size of the debugged component is clearly not a dominating factor on the execution efficiency of Stethoscope.

V. Conclusion

Low-cost networked computing devices get ubiquitous in the world nowadays. While those computers play increasingly important roles, how to ensure them behave correctly remains an open problem. Our work in this paper addresses an important issue, i.e. the runtime applicability and sustainability of WSN debugging. Our proposal, named Stethoscope, incorporates the techniques like Quick Switch and Hooking. The implemented debugger is highly energy-efficient and effective in resolving all kinds of software system requests. In the future, we plan to further investigate the diversity of embedded operating systems and embedded hardware. We will implement Stethoscope on those widely adopted software/hardware platforms, and apply debugging techniques in real system management.

VI. Acknowledgment

This study is supported in part by NSFC under grants No. 61170213, No. 61572277, No. 61202359 and National Science Fund for Excellent Young Scientist No. 61422207.

References

[1] T. He, S. Krishnamurthy, L. Luo, T. Yan, L. Gu, R. Stoleru, G. Zhou, Q. Cao, P. Vicaire, J. A. Stankovic *et al.*, "Vigilnet: An integrated sensor network system for energy-efficient surveillance," *ACM Transactions on Sensor Networks*, vol. 2, no. 1, pp. 1–38, 2006.

[2] "MSP-FET430UIF," http://www.ti.com/tool/msp-fet430uif.

[3] "The *GNU* Project Debugger(GDB)," http://sources.redhat.com/gdb/.

[4] J. Yang, M. Soffa, L. Selavo, and K. Whitehouse, "Clairvoyant: a comprehensive source-level debugger for wireless sensor networks," in *Proceedings of the 5th ACM SenSys*, 2007.

[5] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu, "Finding and reproducing heisenbugs in concurrent programs." in *Proceedings of the 8th USENIX OSDI*, 2008.

[6] R. Sasnauskas, O. Landsiedel, M. Alizai, C. Weise, S. Kowalewski, and K. Wehrle, "Kleenet: discovering insidious interaction bugs in wireless sensor networks before deployment," in *Proceedings of the 9th IEEE/ACM IPSN*, 2010.

[7] N. Kothari, T. Millstein, and R. Govindan, "Deriving state machines from tinyos programs using symbolic execution," in *Proceedings of the 7th IEEE/ACM IPSN*, 2008.

[8] P. Li and J. Regehr, "T-check: bug finding for sensor networks," in *Proceedings of the 9th IEEE/ACM IPSN*, 2010.

[9] T. Sookoor, T. Hnat, P. Hooimeijer, W. Weimer, and K. Whitehouse, "Macrodebugging: Global views of distributed program execution," in *Proceedings of the 7th ACM SenSys*, 2009.

[10] V. Krunic, E. Trumpler, and R. Han, "Nodemd: Diagnosing node-level faults in remote wireless sensor systems," in *Proceedings of the 5th ACM MobiSys*, 2007.

[11] V. Sundaram, P. Eugster, and X. Zhang, "Efficient diagnostic tracing for wireless sensor networks," in *Proceedings of the 8th ACM SenSys*, 2010.

[12] M. Khan, H. Le, H. Ahmadi, T. Abdelzaher, and J. Han, "Dustminer: troubleshooting interactive complexity bugs in sensor networks," in *Proceedings of the 6th ACM SenSys*, 2008.

[13] B. Chen, G. Peterson, G. Mainland, and M. Welsh, "Livenet: Using passive monitoring to reconstruct sensor network dynamics," in *Proceedings of the 4th IEEE DCOSS*, 2008.

[14] L. Luo, T. He, G. Zhou, L. Gu, T. Abdelzaher, and J. Stankovic, "Achieving repeatability of asynchronous events in wireless sensor networks with envirolog," in *Proceedings of the 25th IEEE INFOCOM*, 2006.

[15] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin, "Sympathy for the sensor network debugger," in *Proceedings of the 3rd ACM SenSys*, 2005.

[16] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler, "Marionette: using rpc for interactive development and debugging of wireless embedded networks," in *Proceedings of the 5th IEEE/ACM IPSN*, 2006.

[17] Q. Cao, T. Abdelzaher, J. Stankovic, K. Whitehouse, and L. Luo, "Declarative tracepoints: a programmable and application independent debugging system for wireless sensor networks," in *Proceedings of the 6th ACM SenSys*, 2008.

[18] "mspgcc," http://sourceforge.net/apps/mediawiki/mspgcc/.

[19] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr, "Efficient memory safety for tinyos," in *Proceedings of the 5th ACM SenSys*, 2007.

[20] P. Levis, N. Lee, M. Welsh, and D. Culler, "Tossim: Accurate and scalable simulation of entire tinyos applications," in *Proceedings of the 1st ACM SenSys*, 2003.

[21] Y. Chen, O. Gnawali, M. Kazandjieva, P. Levis, and J. Regehr, "Surviving sensor network software faults," in *Proceedings of the 22nd ACM SIGOPS*, 2009.

[22] A. Chlipala, J. Hui, and G. Tolle, "Deluge: data dissemination for network reprogramming at scale," *University of California, Berkeley, Tech. Rep*, 2004.

[23] "Atmega128L microcontroller," http://www.atmel.com/images/doc2467.pdf.

[24] "Texas Instruments mixed signal microcontroller," http://www.ti.com.

[25] "TinyOS," http://www.tinyos.net/.

[26] Y. Liu, Y. He, M. Li, J. Wang, K. Liu, and X. Li, "Does wireless sensor network scale? a measurement study on greenorbs," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 10, pp. 1983–1993, 2013.