SOFTWARE AND HARDWARE APPROACHES FOR RECORD AND REPLAY OF WIRELESS SENSOR NETWORKS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Matthew Edward Tan Creti

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2015

Purdue University

West Lafayette, Indiana

This thesis is dedicated to my grandfather, Dr. Edward Eaton Mason, who is an inspiration to achieve great things.

To my grandmother, Dordana Fairman Mason, who looked after me during my undergraduate years.

To my father, who gave me a love of learning.

To my mother, who who gave me a love of music.

ACKNOWLEDGMENTS

I would like to thank the many people who have contributed to this thesis and to the research I have performed during my graduate studies. Without their support, this work would not have been possible.

Many thanks to my advisor, Professor Saurabh Bagchi, for giving me the opportunity and freedom to explore new ideas.

My gratitude to the members of my Program Committee, all of whom with which I have had the privilege of collaborating on research, who are Professors Vijay Raghunathan, Zhiyuan Li, and Yung-Hsiang Lu.

Thanks goes to the wonderful collaborators who I have worked with during my graduate studies, besides those mentioned above they include Dr. Vinaitheerthan Sundaram, Professor Patrick Eugster, Dr. Mohammad Sajjad Hossain, Matthew Beaman, Patrick Hurley, Dr. Partha Pal, and Amy Fedyk.

Thanks goes to Dr. Henry Medeiros and Anderson Nascimento for alerting us to unusually long lived routing loops observed in CTP, and Spensa Technologies, Inc. for providing us access to their testbed.

This material is based upon work supported by the National Science Foundation under Grant Nos. CNS-0953468, CNS-0716271, ECCS-0925851, and CNS-0834529. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

μC	Microcontroller
EEM	Enhanced Emulation Module
FPGA	Field-Programmable Gate Array
JTAG	Joint Test Action Group
MAB	Memory Address Bus
MDB	Memory Data Bus
OCDM	On Chip Debug Module
PC	Program Counter
PLL	Phase-Locked Loop
TDB	Telos Debug Board
TI	Texas Instruments
WSN	Wireless Sensor Network

ABSTRACT

Tan Creti, Matthew Edward Ph.D., Purdue University, August 2015. Software and Hardware Approaches for Record and Replay of Wireless Sensor Networks. Major Professor: Saurabh Bagchi.

Wireless Sensor Networks (WSNs) are used in a wide variety of applications including environmental monitoring, electrical grids, and manufacturing plants. WSNs are plagued by the possibility of bugs manifesting only at deployment. However, debugging deployed WSNs is challenging for several reasons—the remote location of deployed nodes, the non-determinism of execution, and the limited hardware resources available. A primary debugging mechanism, record and replay, logs a trace of events while a node is deployed, such that the events can be replayed later for debugging. Existing recording methods for WSNs cannot capture the complete code execution, thus negating the possibility of a faithful replay and causing some bugs to go unnoticed. Existing approaches are not resource efficient enough to capture all sources of non-determinism. We have designed, developed, and verified two novel approaches to solve the problem of practical record and replay for WSNs.

Our first approach, Aveksha, uses additional hardware to trace tasks and other generic events at the function and task level. Aveksha does not need to stop the target processor, making it non-intrusive. Using Aveksha we have discovered a previously unknown bug in a common operating system.

Our second approach, Tardis, uses only software to deterministically record and replay WSN nodes. Tardis is able to record all sources of non-determinism, based on the observation that such information is compressible using a combination of techniques specialized for respective sources. We demonstrate Tardis by diagnosing a newly discovered routing protocol bug.

1. INTRODUCTION

1.1 Motivation

Debugging is one of the fundamental tools for identifying software defects ("bugs"). Debugging is particularly relevant in wireless sensor networks (WSNs), as these are susceptible to unpredictable runtime conditions. Indeed, programmers of WSNs use tools such as simulators [1, 2], safe code enforcement [3], and formal testing [4] prior to deployment of an application in the field, yet exhaustive testing of *all* conditions in the lab is infeasible, because WSNs are deployed in austere environments whose behavior cannot be easily duplicated in a laboratory.

Debugging is often performed in a cyclic process of repeatedly executing a program and tracking down bugs. In WSNs, cyclic debugging can be lengthy and laborious:

- Nodes are often not easily physically accessible, meaning that the programmer must rely on low-power wireless links to painstakingly collect any data of interest.

- There may not be enough information available to immediately diagnose a bug, so the network must be wirelessly reprogrammed with code to collect additional debugging data. This can take minutes, and waiting for the bug to resurface may also take some time.

Once a bug fix is applied, the network is again wirelessly reprogrammed, and further monitoring is required to determine that the bug has been successfully fixed. The cyclic debugging fix-and-test approach thus becomes particularly laborious in this environment.

Record and replay can potentially make the process of cyclic debugging less tedious. With record and replay debugging, program execution is recorded on-line

and then reproduced off-line. Record and replay cuts down on the cyclic process of debugging by capturing a program's execution such that it can be *deterministically* reproduced and carefully examined off-line, perhaps in a hunt for elusive bugs [5]. In addition, in WSNs, the recording can happen on the nodes and the replay and debugging can happen on the relatively resource rich desktop-class machines in the lab. The typical workflow for record and replay in WSNs is that during normal execution of a deployed WSN, the nodes execute instrumented binaries that record a trace of all sources of non-determinism to flash. The trace can then be brought back to the lab for off-line replay. This can be done either through wireless data collection or by physically accessing a node. In the lab, the recorded data is fed into an emulator, which deterministically replays the node's execution. The replay allows a developer to examine the program's execution, including its interactions with the environment, at any arbitrary level of detail, such as through setting breakpoints or querying the state of memory. Such replay helps the developer identify the root cause of bugs encountered in the field.

Based on the experiences shared in "The Hitchhiker's Guide to Successful Wireless Sensor Network Deployments," Barrenetxea et al., provide the following principle [6].

> *Do not throw away even a single byte of raw data.*

Keeping all of the data collected from a WSN is valuable advice, because it is not always possible to foresee how collected sensor data might be used in the future. Data storage is generally inexpensive, while having the reproduce experiments can be time consuming. The data should be available in its raw form so that it is possible for new data transformations to be applied. This may become necessary if a defect was discovered in the original data processing performed *in situ*. Inspired by the guides advice, the work of this thesis is guided by a similar principle.

> *Maintain as much detailed information as possible about the execution of WSNs in deployment.*

The meaning of the statement is that the exact execution of nodes (e.g., their instructions and memory state) should be recorded such that it can be reproduced at a later time. This leads to two questions: "Why keep such detailed information?" and "How is recording the complete execution of a node feasible given the resource constrains of sensor nodes?"

Why keep such detailed information? Just like sensor data is valuable for future data mining, execution traces of nodes are valuable for discovering defects in deployed WSNs. Many unexpected defects manifest themselves only in deployment. However, it is difficult to obtain detailed information on the execution of nodes that have been deployed, due to their remoteness and their resource constrains. Ideally we would like to be able to replay every memory state and instruction execution, because this makes reproducing *a*ll software bugs possible. In Chapter 3 we demonstrate a software approach called Tardis which can do just that.

How is recording the complete execution of a node feasible given the resource constrains of sensor nodes? In an effort to reduce the cost, size and energy consumption of sensor nodes, the main processor and non-volatile storage are heavily constrained in WSNs. The main processor is typically a microcontroller (μC) which may be limited to a few MHz and RAM in the range of tens of KBs. Non-volatile storage is usually a flash chip which may contain anywhere from a MB to a few GB of storage. Radios typically only have kilobytes per second of bandwidth. Given this it does not seem feasible to record and replay executions down to the instruction level, and to do so in a manner that does not interfere with the real-time constrains of sensor nodes. In Chapter 2, we present a hardware approach that is able to generate traces at the granularity of every function call using Commercial Off-The-Shelve (COTS) components already available in most sensor nodes. In this work hardware based parallelization is the key to achieving high granularity tracing, without interfering with the execution of the node being traced. In Chapter 3, we present a software-only based solution that uses carefully chosen compression techniques and records

only the bare minimum non-deterministic data such that a deterministic replay of the execution can be produced.

1.2 Outline

In Chapter 1 we introduce the motivation for this thesis and list the papers that have resulted from this research. In Chapter 2 we present Aveksha, a hardware based approach to trace tasks and other generic events at the function and task level. The hardware and firmware design for Aveksha is presented in section Section 2.1. Aveksha has different operating modes that determine what type and granularity of trace can be collected as explained in Section 2.2. The evaluation of Aveksha's performance and functionality along with a newly discovered bug in a common open source operating system is presented in Section 2.3. Section 2.4 describes work related to Aveksha. In Chapter 3 we present TARDIS, a software based approach to deterministically record and replay WSN nodes. Section 3.1 introduces the challenges faced by TARDIS and the the approach and contributions of TARDIS. The design of TARDIS is given in Section 3.2 followed by an evaluation of the performance and functionality of TARDIS in Section 3.4. A discussion of limitations and future work is provided in Section 3.5. Work related to TARDIS is described in Section 3.6. The conclusions are provided in Chapter 4.

1.3 Published Work

Parts of this thesis have been presented and published in the proceedings of peer reviewed conferences.

- Matthew Tancreti, Mohammad Hossain, Saurabh Bagchi, and Vijay Raghunathan, "Aveksha: A Hardware-Software Approach for Non-intrusive Tracing and Profiling of Wireless Embedded Systems," in *Proceedings of the 9th ACM*

Conference on Embedded Networked Sensor Systems, SenSys '11, 14 pages, Seattle, Washington, November 1-4, 2011. **(Winner of the best paper award.)**

- Matthew Tancreti, Vinaitheerthan Sundaram, Saurabh Bagchi, and Patrick Eugster, "TARDIS: Software-Only System-Level Record and Replay in Wireless Sensor Networks," in *Proceedings of the 14th ACM/IEEE Conference on Information Processing in Sensor Networks*, IPSN '15, 12 pages, Seattle, Washington, April 14-16, 2015.

2. HARDWARE BASED TRACING AND PROFILING OF WIRELESS SENSOR NETWORKS

In the first approach to detailed record and replay, we present Aveksha [7]. Aveksha is a system based on an insight that most processors, including low-cost embedded processors, offer visibility into their internal workings through an On-Chip Debug Module (OCDM), whose signals are exposed through a standard JTAG interface. This interface has been used by embedded system engineers primarily for interactive debugging, such as single stepping, showing values of registers, *etc.* We show how this visibility, together with the fact that most OCDMs provide a general-purpose method of setting triggers, can be leveraged to perform automated tracing in a deployed setting.

In this work we would like to have visibility at a fine granularity - both spatially and temporally. Spatially fine visibility implies that it should be possible to trace *individual* events of interest as opposed to only bursts of events (clearly, tracing *every* event is likely to be prohibitive) and it should be possible to trace performance and energy at fine code regions, such as a function or a task (using TinyOS terminology). This is desirable because the fine region of code can then be debugged if it is determined through performance profiling that this region is causing a performance bottleneck, through energy profiling that it is consuming unexpectedly large amounts of energy, or through record and replay that it is the source of a bug. Temporally fine visibility implies that it should be possible to do the tracing with a high sampling frequency. Clearly, the two dimensions are not independent. In order to trace small regions of code in a loop, it is necessary to be able to trace at a fine temporal granularity.

While the problem motivation laid out above has been clear to researchers for quite some time [8], it has proved very difficult to provide a solution for low-cost embedded

wireless nodes that can operate at a large deployed scale. The first line of attack has been to provide pure software solutions [9–11]. Such solutions have perturbed the application too much to be useful for many of the use cases indicated above. For one, they change the timing behavior enough that some bugs get suppressed. Else, they cause such a large slowdown in the application execution that it is not possible to employ them in a deployed setting. To get around this problem, a recent software solution [11] has focused on a specific kind of tracing (control flow tracing) and intelligent static analysis and runtime trace collection, compression and storage. Thus, it addresses one of the above usage scenarios. The second line of research has developed hardware solutions for subsets of the usage scenarios laid out above. For example, [12] has developed a dedicated integrated circuit, implemented using an FPGA, that is tightly integrated with the host processor and its peripherals and can measure energy drawn accurately at millisecond resolution. Quanto [13] is a solution that de-emphasizes sophisticated hardware design. Instead, it measures energy at the node level, uses indication from device drivers about changes in power state, and performs causality tracking to pin down energy usage due to individual activities. Thus, Quanto is a hardware-software solution, and like all prior solutions that have a software part, is OS-specific (in this case, TinyOS).

A high-end hardware solution for tracing the execution on an embedded processor is provided by solutions such as Green Hills Software's SuperTrace probe and TimeMachine tools [14]. These solutions can collect fine-grained trace data from nearly all 32-bit and 64-bit processors, even those without integrated trace hardware. Unfortunately, such solutions are very expensive in dollar terms (*e.g.,* the SuperTrace probe and TimeMachine tools together cost almost $15,000) and are not available for the low-end embedded processors that are commonly used in embedded wireless nodes.

We develop a debug board formed of standardized components – a microcontroller unit (μC), which in our case happens to be the same as the application processor, MSP430F1611 from Texas Instruments, and an Actel FPGA, both of which interact

with the OCDM on the application processor over the JTAG interface. We refer to our debug board as the Telos Debug Board (TDB) because it is intended to be used with the Telos wireless sensor node (however, our solution is not restricted to the Telos and can easily be adapted to other embedded platforms based on the MSP430 microcontroller, and with some effort to other embedded platforms). The MSP430 OCDM (also referred to by the microcontroller datasheets as the Enhanced Emulation Module or EEM) allows Aveksha unprecedented visibility into the state of the application processor. Further, the OCDM has a small circular buffer where events of interest can be stored and subsequently drained by the FPGA on the TDB. The triggering mechanism of the OCDM is very flexible and is therefore attractive for Aveksha. For example, the OCDM can be triggered to indicate when the application processor has accessed a certain memory region or a certain peripheral device, such as a sensor. We find that the triggering mechanism can be combined with thoughtful design to trace all the events of interest for our three usage scenarios – performance profiling, energy profiling, and record-and-replay.

One challenge that we face, and resolve partially, is the need to do real-time tracing, *i.e.,* without interrupting the application processor. Aveksha is able to achieve this when the rate of events that it has to trace does not exceed some bound, which depends on the mode of tracing it uses. Aveksha operates in one of three modes: *breakpoint*, *watchpoint*, and *program counter (PC) polling*. Breakpoint is a baseline and we use it for demonstrating some functionality of the TDB. It is intrusive and, therefore, does not meet our solution requirements. The *watchpoint* mode has Aveksha set triggers, where each trigger unambiguously maps to an event of interest (such as when a sensor is read). When a trigger fires, the application processor is not stopped, but the state is dumped to a buffer on the OCDM, which is then emptied out by Aveksha. This is a rate-limited operation and if events of interest happen with a high enough frequency, the buffer overflows and Aveksha misses some events of interest. In the *PC polling* mode, the TDB tracks the program counter values of the application processor without interrupting it. Then, it processes the PC values

to determine events of interest, such as when control flow has entered a particular function. These three modes reveal different tradeoffs in terms of intrusiveness, the flexibility in defining which events to collect, and the rate at which collection can be done.

The Aveksha approach is:

1. The first technique for non-intrusive tracing of a wide variety of events, including arbitrary user-defined events, in embedded wireless nodes. We motivate the events of interest from three well-accepted usage scenarios.

2. A tracing technique that is agnostic to the operating system, compiler infrastructure, or language in which the application is implemented.

3. Implemented in hardware that is built using off-the-shelf components and requires little effort in integrating with the application board, which is modified only very slightly for enabling the tracing.

4. Suitable for deployment at a large scale because it is low cost, can operate on battery power, and extracts program information directly from the application processor.

Our solution also has some limitations. In brief, the TDB is a relative energy hog itself, drawing about the same power as the application processor board. Its ability to keep pace with events is exceeded if a burst of 8 events happens within a window smaller than 976 clock cycles (in the watchpoint mode) or events happen more frequently than 7 clock cycles (in the PC polling mode).

2.1 TDB Hardware and Firmware

In this section, we present the design of the Telos Debug Board (TDB), which provides execution tracing and energy monitoring of the Telos Rev B mote. An μC and an FPGA provide the programmability of the TDB.

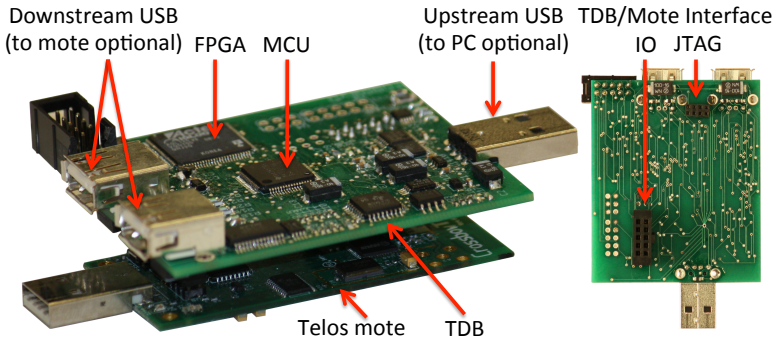


Fig. 2.1.: The Telos Debug Board (TDB) with a Telos Rev B mote underneath, and the underside of the TDB (right).

Terminology: We lay out some terminology that we will use through the rest of the paper. We wish to monitor the execution of the *application processor* that is part of an *application processor board*. The application processor board, which we sometimes also refer to as the *mote*, has various peripherals such as sensors and the JTAG interface in addition to the application processor. We refer to the hardware board that is a part of our solution as the Telos Debug Board, while the entire hardware-firmware that forms our solution is called Aveksha.

Figure 2.1 shows a photograph of the TDB with a Telos mote attached underneath. The MCU, FPGA, and multiple USB ports on the TDB are highlighted. The figure also shows the JTAG and IO connections between the TDB and the mote. The TDB is designed so that it can be deployed in the field, connected to a mote. In this mode of operation, a battery powers the TDB, which in turn provides power to the mote. As a secondary mode of operation, the TDB can also stream logged events directly to a USB host such as a laptop. This is useful for in-lab debugging.

2.1.1 Energy Monitoring

Energy is a key concern for sensor networks, because motes must operate unattended on battery power for long periods of time. When optimizing an application to reduce energy consumption, it can be useful to observe how much energy is consumed

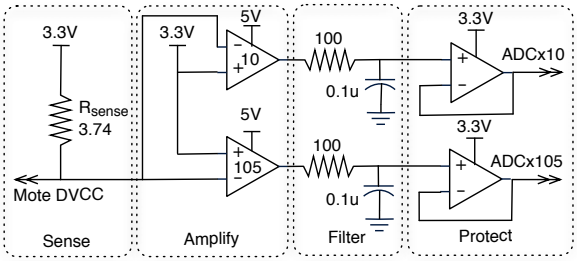


Fig. 2.2.: Simplified schematic of the energy monitoring circuit of the TDB.

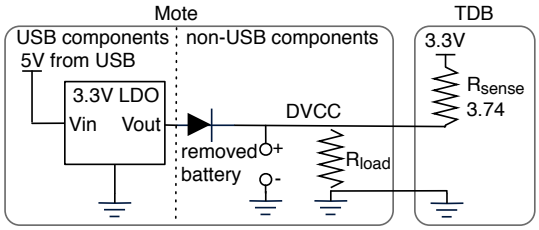


Fig. 2.3.: Simplified schematic of powering the mote through the TDB. The mote's battery should not be connected.

in different states. The TDB can measure and log the power consumed by the mote, which can then be correlated to different operational states of the mote.

The standard method for measuring energy consumption is by monitoring the voltage over a sense resistor. The sense resistor (R_{sense}) is placed in series with the load being measured. The voltage (V) across R_{sense} is sampled and the load's current draw is calculated by $I = V/R_{sense}$. The supply voltage (V_{supply}) can then be used to find the power being drawn by the load with $P = I * V_{supply}$. The power samples can then be integrated over time to determine energy consumption.

The challenges presented in monitoring energy in sensor networks is the wide dynamic range of power draw of a mote and the rapid changes in power draw. For example, a mote may draw only tens of μA in sleep mode and as much as $30mA$ when fully active. It may not be sufficient to ignore the small power draw when the mote is in low power mode, because typical sensor network applications spend long periods of time in the low power state while only waking for short periods of time. Further,

the change in the current draw when the mote transitions from one state to another is rapid.

To meet these challenges, we use two instrumentation amplifiers to amplify the voltage across R_{sense} by a gain of 10 and 105, as shown in Figure 2.2. Because R_{sense} is placed on the high side, the amplifiers need to be supplied with a voltage larger than 3.3V, in this case 5V. The output of these amplifiers is fed into an RC low-pass filter to avoid interference from high frequency components. The cut off frequency is about 16kHz. This value is justified in the design of the SPOT energy meter, based on an observed significant drop in energy content above this frequency [15]. Another pair of amplifiers with unity gain is used to protect the ADCs which cannot tolerate more than 3.3V. Two ADC channels of the MCU on the TDB sample the x10 and x105 lines at 20KHz. The ADCs are 12-bit and sample against a reference voltage of 2.5V. This gives ADCx10 a resolution of $61\mu V$ across R_{sense} which is equivalent to $16.3\mu A$ of current draw, and a maximum reading of 250mV or 66.8mA, which is more than the mote's maximum current draw of 30mA. The ADCx105 gives a resolution of $5.81\mu V$ across R_{sense} which is equivalent to $1.55\mu A$ of current draw, and a maximum reading of 23.8mV or 6.37mA.

As shown in Figure 2.3, R_{sense} is placed between the TDB's 3.3V supply and the mote's DVCC line, while the ground lines of the mote and the TDB are shared. This configuration is known as high side sensing. One advantage of high side sensing is that the ground plane of the mote and the TDB are shared. Another advantage is that a Zener diode built into the Telos achieves isolation between the USB components and the non-USB components, and thus allows us to capture the current draw of only the non-USB components, even when the mote is connected to a USB host. The diode has a forward bias of about 360mV, meaning that as long as the voltage drop across the sense resistor remains below 360mV, all current drawn through R_{load} will be from the TDB. R_{sense} is chosen sufficiently low such that this will happen even at maximum power draw by the mote. The maximum current of the mote is 30mA which would result in a voltage drop across the sense resistor of 112mV.

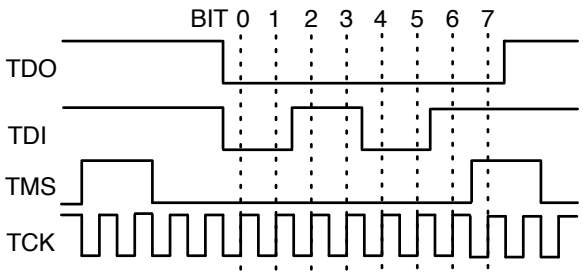


Fig. 2.4.: Timing example for shift IR. The byte 0xCC is shifted in on TDI while 0x00 is shifted out on TDO.

To account for amplifier offset we use a switch between R_{sense} and DVCC that allows 3.3V to be temporarily placed at both ends of R_{sense} in a manner similar to [15]. The MCU has an ADC buffer of 16 samples, that are filled by DMA to reduce overhead. The x10 and x105 amplified signals are sampled alternately at a rate of 40ksps, to achieve an effective sampling rate of 20ksps. When the ADC buffer is full, an interrupt service routine sums up the samples in the buffer. It is desirable to use the ADCx105 reading due to its greater current resolution, unless there has been an overflow in its reading. The ADCx10 value is used to determine if the ADCx105 has had an overflow.

Using the MCU's ADC reduces system cost and complexity. However, it does have the disadvantage of introducing a delay between an event occurring on the application processor, and the the MCU recording the energy. The average delay is 234μs. 30μs is taken by the FPGA to read the event from JTAG, while the rest is mostly due to the time required for the MCU to sum the samples of the 16 entry ADC buffer. An alternative design would have the FPGA poll a separate ADC. This would reduce the delay to below the 50μs (20KHz) sample period of the ADC.

2.1.2 JTAG Background

The application processor contains an on-chip debug module. This module can be used to emulate the processor (directly control the processor operations) and it can

execute breakpoints and watchpoints when certain conditions of the data and address buses are met. A breakpoint halts execution of the processor, while a watchpoint records the contents of the data and address bus into an 8 entry circular buffer. The OCDM is implemented as a state machine that is controlled via the standard JTAG protocol.

JTAG uses four lines: data output to host (TDO), data input to target (TDI), mode select (TMS), and clock (TCK). JTAG shifts frames of data into and out of the OCDM and that changes the state of the OCDM. There are two basic shift modes: an 8-bit instruction register (IR) shift and an n-bit data register (DR) shift. The TMS line selects between IR or DR at the start of a shift based on the number of TCK rising edges for which it remains high. For example, in Figure 2.4, TMS remains high for two rising edges which selects the IR mode, while one rising edge would select the DR mode. The number of bits shifted in an n-bit DR shift is determined by TMS being high a second time during the shift of the last bit. Bits are shifted from the mote to the TDB on the TDO line and from the TDB to the mote on the TDI line. Although the JTAG protocol is standard, the sequence of instructions that must be shifted into the OCDM on the MSP430 is proprietary. We have reverse engineered these control sequences and used them in Aveksha to determine what command sequences must be sent to the OCDM for the application to enter a breakpoint, to set a watchpoint, or to enable PC polling.

2.1.3 Hardware Architecture

As shown in Figure 2.5, the TDB consists of a USB hub, a USB to UART adapter, an MCU, and an FPGA. The USB components are primarily for use in a lab environment and provide reprogramming, control, and streaming of log data. The USB hub has 1 upstream port and 3 downstream ports. The upstream port is used to access the debug board from a PC. One of the downstream ports is permanently connected to a USB to UART adapter that provides reprogramming and data transfer to and

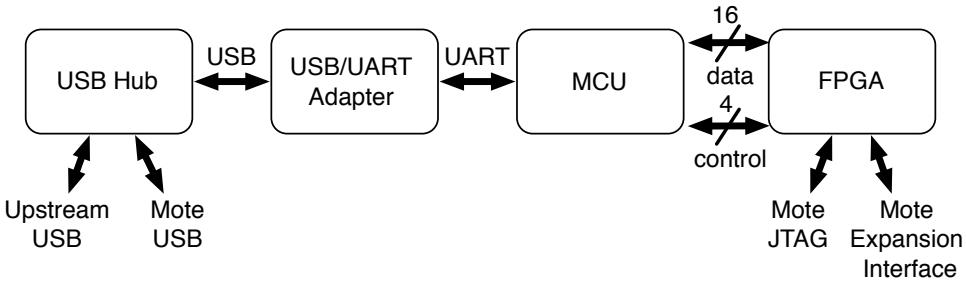


Fig. 2.5.: Hardware architecture of the TDB.

from the MCU. The second downstream port is available for connection to the USB port on the mote. This is useful in a lab or testbed deployment where access to the mote's USB port is desired. The final downstream port is available for future use, and we envision it being used in testbed deployments to daisy chain several TDBs together.

The Actel IGLOO nano FPGA interfaces with the mote's JTAG and expansion interfaces. The expansion interface of the mote provides access to the some of the its UART, I2C, ADC, and GPIO peripherals. It was necessary to use an FPGA to control the mote's JTAG to be able to poll at a sufficient frequency to keep up with the events we want to observe. For some operations (such as PC polling) we have found the need to drive the clock line of the JTAG up to 24MHz. A software implementation would require a processor that operates at several times that speed. Additionally, there is the problem of processing the collected JTAG data to determine what should be logged. The FPGA allows pipelining of JTAG control and data processing, so that the polling loop never waits for data processing.

The MCU performs tasks that are less time critical and better suited to software. For example, initialization of the mote for debugging, reading the contents of the mote's program memory, and disassembly of the mote's program code are performed by the MCU. Using an MCU also makes adding functionality to the debug board easier because the MCU can be reprogrammed over USB. To simplify programming,

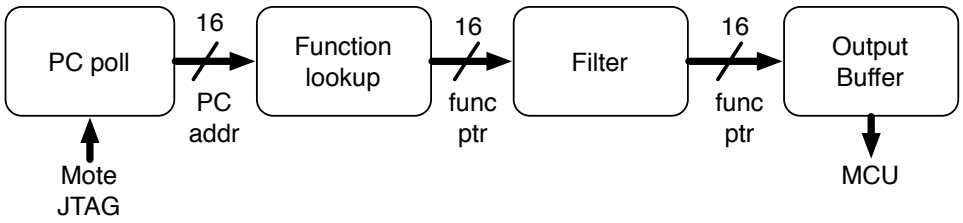


Fig. 2.6.: FPGA pipeline in PC polling mode.

the MCU used on the TDB is an MSP430 processor that is identical to the one used on the Telos mote. It operates at 8MHz.

In the current prototype, the TDB logs are streamed over USB. It would also be possible to add some Flash memory to act as a circular buffer as was done in FlashBox [16]. Our maximum reliable streaming throughput over USB is 1Mbps. This is limited by the USB 1.0 hub and adapter, which have a theoretical throughput of 1.5Mbps. Moving to USB 2.0 would boost the USB throughput to 480Mbps, which would make the MCU the bottleneck in streaming logs. However, we have found 1Mbps to be sufficient in all of our experiments as long as buffers are added in the MCU and FPGA to absorb short bursts of data to be logged.

2.1.4 Firmware Architecture

The firmware of the TDB consists of C code for the MCU and Verilog code for the FPGA.

Firmware on the MCU: The MCU is responsible for initialization tasks. When the TDB is first connected to the mote, the MCU sets the FPGA into a mode where the MCU can directly control the JTAG lines connected to the mote. Through JTAG commands, the mote is put into a halt state and the program memory of the mote is read. A simple disassembly of the program is performed, where the start of every function block is discovered by examining the destination of every call instruction in the code. This approach will not reveal functions that are called only by function pointer, however such functions are not common, and if they must be debugged then

the symbol table generated from a compiler can be loaded. The resulting table of function blocks is then programmed into the FPGA's RAM for use by the *function lookup module* – a module that takes an address as input, and outputs the function block containing that address. Once the table is loaded, any watchpoint trigger can be set on the mote. What triggers will be set will depend on the goal of the tracing. We explain in Section 2.2 the complete list of triggers that can be supported. Finally, the MCU sets the FPGA to either the PC polling or the watchpoint mode and resumes execution on the mote. Thus, the MCU on the TDB functions as an orchestrator, but leaves the core functionality for the FPGA.

The main advantage of reading and disassembling the mote's program memory when the TDB is connected to the mote is that the TDB need not be aware *a priori* of what application the mote is running. This process is independent of the compiler or operating system used by the mote. In addition to finding the entry point of every function, the disassembly of the code is also used to record the start and return of every interrupt service routine, the address of every function call and return, and the addresses of special `nop` instructions (e.g. `MOV R4, R4`), that are used as trigger markers in the code.

Firmware on the FPGA: The FPGA is responsible for controlling the mote's OCDM through JTAG. The FPGA polls the OCDM to to detect the occurrence of any events of interest. Following each iteration of the polling loop, processing may need to be performed to decide whether or not the polled data should be logged. For example, with PC polling, a log entry should be generated when the polled PC value falls into the address range of a new function. To prevent a slowdown in polling, processing is pipelined. Figure 2.6 shows the PC polling pipeline. At the end of each PC poll, the PC address is passed to the function lookup module. The function lookup module contains a table in RAM of the start address of every function block. A binary search is performed on the table to find the start address of the function block that corresponds to the polled PC address. The function table capacity is 1024 function pointers, so the lookup completes in at most $log_2(1024) = 10$ reads from

RAM. Lookup is not a bottleneck, because the FPGA is internally clocked at 48 MHz, which allows 76 clock cycles per PC poll. After the correct function pointer is discovered, it is passed to a filtering module. This module decides to log the function pointer only if it is different from the last logged function pointer. Finally, function pointers that are to be logged are passed to a FIFO output buffer maintained in the FPGA. The MCU on the TDB reads this buffer and logs the data either in local Flash memory, or if required, streams the log entries to a host machine over USB. The buffer is necessary because the MCU performs other functions, such as energy monitoring, and may not be able to read a value to be logged in the time it takes to perform a single PC poll. The buffer also absorbs peaks in the rate of new functions being invoked. We have observed a buffer size of 256 to be sufficient to absorb all peaks in the programs that we have monitored.

2.2 Using the Architecture for Tracing and Profiling

There are three modes that Aveksha can operate in while monitoring application execution, namely *Breakpoint mode*, *Watchpoint mode*, and *PC Polling mode*. Depending on the mode of operation, Aveksha interacts with the OCDM on the application processor in different ways. Therefore, these modes have different tradeoffs in terms of the level of intrusiveness to the application (breakpoints are the most intrusive), the flexibility offered in terms of the kinds of events that can be observed (watchpoints are the most flexible), and the speed of event logging (PC polling is the fastest). Before we describe the three modes of operation, we discuss the kinds of triggers that Aveksha can set for observing events of interest on the application processor.

2.2.1 Types of Triggers Available

The OCDM on the application processor allows us to set 8 concurrent triggers for detecting events of interest. Although this number may, upon first glance, seem

Table 2.1.: Types of triggers available for monitoring events.

Event	Condition	# Triggers
Function call	MDB-F==0x12B0	1
Function return	MDB-F==0x4130	1
Interrupt	MAB-R≥0xFFE0	1
Interrupt return	MDB-F==0x1300	1
Peripheral read	0x0010≤MAB-R≤0x01FF	2
User defined	MDB-F==0x4404	1

insufficient to create a complete profile of an application, that is not the case because the MSP430 offers far more advanced triggers than just the program counter (PC) reaching a particular value. For example, a trigger can compare the Memory Data Bus (MDB) or the Memory Address Bus (MAB) to a set value or range of values. Additionally, the trigger can be restricted to be active only during an instruction fetch (F), a memory read instruction (R), or a memory write instruction (W). This gives us great flexibility in using these 8 concurrent triggers to capture all our events of interest.

All of the triggers that we use in this paper are listed in Table 2.1. The notation used for specifying the condition that the value on the Memory Data Bus equals `0x12B0` on an instruction fetch is given by: `MDB-F==0x12B0`. This particular trigger will fire for every function call because `0x12B0` is the machine code for a function call instruction. Similarly, the machine code for the return instruction from a function call is `ret=0x4130`. Therefore, the trigger `MDB-F==0x4130` will trigger on all function call return events. A call to an interrupt can be detected with the trigger `MAB-R≥0xFFE0`. The interrupt vector table is located between address `0xFFE0` and the end of the address space at `0xFFFF`. Every time an interrupt is to be serviced, the processor reads the interrupt vector table to determine the address of the interrupt service routine

that corresponds to the interrupt being serviced. Interrupts have their own return instruction (`reti=0x1300`) that can be monitored with the trigger `MDB-F==0x1300`.

The compound trigger `0x0010≤MAB-R≤0x01FF` will fire for every read to memory between addresses `0x0010` and `0x01FF`. A compound trigger, such as the above, that contains two conditions is made by joining two triggers together, and uses 2 of the 8 available trigger entries. In the MSP430, the peripherals are all memory mapped to addresses between `0x0010` and `0x01FF`. The peripherals include any sensors that may be attached to the application processor. For purposes of deterministic record and replay, it is important to track what sensor values are read. This can be done by using the trigger `0x0010≤MAB-R≤0x01FF`, which captures a read from the memory-mapped peripheral portion of memory. When a trigger is fired, the OCDM stores the values of the MAB and the MDB to the 8-entry circular buffer. The stored MDB will contain the value that was read from the peripheral.

While functions and interrupts are interesting points for monitoring, we would like even more flexibility to monitor any arbitrary event in the executing application. For example, if we want to monitor the execution of every `task` in TinyOS, we cannot do this with a function call trigger. This is because the `gcc` compiler inlines many of the tasks in the scheduler's `runTask()` function. One solution is to set the `noinline` directive on all task functions. We have verified that this works, however, this is unsatisfactory because it sacrifices the efficiency gains obtained due to function call inlining. A less costly solution is to trigger on a `nop` instruction. However, the MSP430 does not have an explicit `nop` instruction. Instead, compilers emulate this instruction by using a 1 cycle instruction that has no direct effect and no side effect on status or mode bits – specifically, `gcc` uses the instruction `MOV R3, R3` to emulate a `nop`. There are three possible 1 cycle instructions that meet the requirements for no effect or side effect: (`MOV Rn, Rn`), (`BIC #0, Rn`), and (`BIS #0, Rn`). With 16 registers available on the MSP430, this gives us 48 possible choices for an emulated `nop`. We can use different application-specific meanings for each emulated `nop` instruction to monitor 48 arbitrary events of interest. For our purposes we choose just one, (`MOV`

`R4, R4`), which translates to the machine code `0x4404`, and add an instruction fetch trigger `MDB-F==0x4404`. A programmer can now place the assembly code (`MOV R4, R4`) at arbitrary places in the code to monitor user-defined events of interest, such as the beginning of a task.

2.2.2 Breakpoint Mode

Any of the 8 concurrent triggers available can be set as a breakpoint. When a breakpoint is reached, the application processor halts execution. Aveksha performs a continuous poll of the CPU state of the application processor. When it sees that the CPU is halted, it retrieves the state of the application processor (*e.g.,* the value of the PC) and sends the JTAG command to resume execution.

Table 2.2 shows the speed at which a single poll (or test) of the CPU state can be performed, the time it takes to read the PC register, and the time it takes to resume CPU execution. All of these operations involve shifting values into the instruction register (IR) and data register (DR) of the OCDM. For example, a test of the CPU state requires shifting one IR and one DR, reading the PC register requires shifting 2 IRs and 4 DRs, and resuming the CPU involves shifting 3 IRs and 1 DR. The operations needed for achieving the tasks are not documented and we determined them through reverse engineering TI's IAR debug interface [17]. An IR can be shifted in 15 cycles of the JTAG clock (TCK) and a DR can be shifted in 23 JTAG clock cycles. Table 2.2 shows the times for performing the required shift operations in software with the MCU on the TDB running at 8MHz, and the FPGA implementation. The FPGA is able to operate the JTAG clock (TCK) at 10MHz, which is the fastest we have been able to operate the JTAG reliably for the breakpoint and the watchpoint modes and is the maximum rated speed according to the JTAG specification.

Breakpoints have the advantage that we never miss a trigger firing, because every time a trigger is reached the CPU is halted and control is passed to the TDB. The disadvantage of the breakpoint mode is that we lose the property of non-intrusiveness.

In TinyOS, the MSP430 on the Telos is set by default to operate at 4MHz meaning $1\mu s = 4cycles$. Using the FPGA implementation, the time to poll and resume the application processor is equivalent to 91.2 cycles of the application processor. The application processor has to be halted for at least this time while processing an event.

2.2.3 Watchpoint Mode

Any of the 8 triggers can be set as watchpoints rather than breakpoints. The JTAG interface has an 8-entry circular buffer where memory address bus (MAB) and memory data bus (MDB) are stored when a watchpoint is hit. As indicated earlier in Section 2.2.1, the trigger can be on an instruction fetch, read, or write. Thus, by recording the address bus content on an instruction fetch, it is possible to know the PC value. The most recent entry written to the 8-entry buffer is indicated with a set flag. Aveksha polls the flag of the most recently written entry until it is cleared, indicating that a new entry has been written to the buffer. It continues to read entries until it again reaches an entry that has the last entry flag set.

Watchpoints have the benefit that unlike breakpoints, they are not intrusive to the application. The application does not have to interrupt its execution when a watchpoint trigger is met. However, this also means that there is a threshold for the rate of triggers that Aveksha can keep up with. Beyond this rate, the circular buffer will wrap around and some events of interest will be missed. Based on our empirical measurements (given in Table 2.2), the entire processing for one invocation of of a watchpoint trigger takes $30.5\mu s$, which corresponds to 122 cycles for the application processor at 4 MHz. Thus, as long as we do not have a sustained burst of 8 events of interest within $8 \times 122 = 976$ cycles, the TDB in the watchpoint mode will not miss any event.

2.2.4 PC Polling Mode

The final approach to trace generation, is to forego using triggers entirely, and instead poll the program counter of the application processor. With this approach, each polled PC value is used to determine what section of the code the application processor is executing in. For example, the function lookup described in section 2.1.4 finds the function corresponding to a PC value. The advantage of PC polling is that it is about 19 times faster than the watchpoint mode and hence can keep pace with a higher frequency of events, such as every function transition. From Table 2.2, we see that it takes just $1.6\mu s$ to complete a single PC poll, which corresponds to 6.4 clock cycles of the application processor. This means that it is possible to miss function transitions shorter than 7 clock cycles. However, with most instructions taking more than one cycle, such a short time between function transitions is unusual. A disadvantage of PC polling is that it does not allow for advanced triggers – it only allows reading the PC value. For example, we could not use PC polling alone to watch for a memory read or write to a specific memory location.

2.3 Evaluation

2.3.1 Microbenchmarks

The objective of our microbenchmarking experiments is to evaluate the performance of the building blocks of Aveksha. In particular, we evaluate (a) how many clock cycles it takes for Aveksha to perform event monitoring for each of the three modes – breakpoint, watchpoint, and PC polling, (b) the accuracy of the energy monitoring by comparing it with measurements obtained using a Fluke multimeter and a dedicated power monitor from Monsoon Inc., and (c) the energy consumption of the TDB itself.

Time Taken in Each Monitoring Mode: Ideally, we would like to poll the PC or the watchpoint buffer at a rate sufficient to observe every instruction executed on

Table 2.2.: Time taken, in software and using the FPGA, to perform various operations through JTAG in the breakpoint, watchpoint, and PC polling modes.

Mode	Operation	Software (μs)	FPGA (μs)
Breakpoint	Test	43	3.8
	Read Addr.	140	12.2
	Resume	77	6.8
	Total	*260*	*22.8*
Watchpoint	Test	200	18.3
	Read Addr.	140	12.2
	Total	*340*	*30.5*
PC Polling	Read PC	48	1.6
	Total	*48*	*1.6*

the application processor. Unfortunately, the MSP430's OCDM was designed to be operated with a maximum frequency of 10MHz for the JTAG clock. One exception we have discovered empirically is that PC polling can be reliably clocked at up to 24MHz. Table 2.2 presents the effect this has on the time taken to complete basic polling operations. The software column shows how long operations take if only the MCU on the TDB is being used while the FPGA column represents the time operations take in the current FPGA implementation. The FPGA implementation is limited only by how fast the JTAG clock of the application processor can be reliably driven. The table presents results in μs. TinyOS operates the main clock at 4MHz by default, so 1μs=4 clock cycles.

The breakpoint mode of monitoring comprises a test operation to determine if the application processor has halted (which is done in a loop), a read address phase

Table 2.3.: Accuracy of the current measurements provided by TDB for fixed resistive loads, compared to values computed based on measurements with a Fluke multimeter.

Resistance	Current		Relative Error
(Ohms)	Computed (μA)	TDB (μA)	(Unitless)
179.71	18362.92	18483.56	0.007
218.55	15099.52	15193.74	0.006
560.8	5884.45	5807.06	0.013
991.4	3328.63	3314.46	0.004
4689.2	703.74	674.77	0.041
32610	101.20	92.34	0.088
55220	59.76	52.97	0.114
179360	18.40	16.09	0.125
266750	12.37	16.28	0.316

to collect the instruction address at which the halt occurred, and a resume phase to restart execution. Likewise, the watchpoint mode has a test phase and a read address phase. The test phase here is more complex because it has to test if a new entry has been created in the JTAG circular buffer. In total, a watchpoint poll takes 30.5μs, or 122 cycles of the application processor. For all of the applications that we have experimented with (which are a superset of the ones for which we provide results here), the rate of events, tasks, and application-level functions is lower than the above rate. However, if we include system-level entities (functions, events, and tasks), then this rate is occasionally exceeded. Finally, PC polling only requires a read PC operation that can be performed in under 7 cycles of the application processor.

Accuracy of Power and Energy Monitoring: The objective of this experiment is to see if Aveksha can faithfully monitor the power draw in the static case (using a

fixed resistive load) and when there are spikes in power consumption, which happen commonly in embedded systems, e.g., when the radio switches on. Table 2.3 shows the current consumption reported by the TDB for various resistive loads. For comparison, we measured the value of each resistor using a high-accuracy Fluke multimeter and computed the theoretical current consumption through it. As seen in the table, TDB's current measurement is within 10% of the computed value for current draws of 100 μA or above, while the error goes up for smaller current values. The accuracy of the current measurement can be improved further using techniques (which we have not implemented yet) such as better decoupling of analog and digital components, and use of a ground plane.

We also measured the power consumption reported by the TDB while attached to a Telos mote running the `TestNetworkLpl` TinyOS application. It is important to note that the amplitude of the power consumption trace in this case will have a significant dynamic range due to various components on the mote changing power states during application execution. The TDB measurement of energy draw over a 1 minute period is within 3.2% of that given by a Monsoon power meter [18]. Since the spikes in energy draw are three orders of magnitude higher than the steady state case, this close result can only be achieved because TDB monitors the current spikes faithfully. For comparison, the static energy metering of iCount over a five decade range is accurate to 20% down to about 1 μA [19].

Power Consumption of TDB: It is important that the TDB itself consume a small amount of power, because the typical usage scenario is the TDB coupled to the application processor board when deployed in the field. A large source of energy saving on the application processor comes from entering a low-power sleep state when not in use. With a small modification of the application processor's code, to signal when it is in sleep mode, it is also possible for the TDB to enter a low-power sleep state.

A general purpose pin on the application processor is used to signal to the TDB when to enter sleep mode. The sleep signal is connected to the Flash Freeze pin on

the FPGA. Flash Freeze is a feature of Actel's IGLOO nano FPGAs, that allow them to enter a low power state while still holding output pins at their previous state. Because the JTAG controlled OCDM is implemented as an externally clocked state machine, as long as the JTAG I/O lines remain static while in sleep mode, the OCDM will have the same state upon wakeup. The FPGA is able to recover from sleep mode within 1 μs.

We have implemented the necessary code to signal sleep mode in TinyOS. Only system code is affected, the application does not change. In total, we added only 10 lines of code to TinyOS. First, we initialized the general purpose pin `GIO0` for output mode in `MotePlatformC`. Then we modified the macro `TOSH_SIGNAL`, which is used to create interrupt handlers, such that the beginning of every interrupt sets `GIO0` to indicate a transition to the awake state. Finally, we modified `McuSleepC` to clear `GIO0` just before transitioning to sleep.

The TDB consumes 55mW when active. The current implementation does not have the ability to disable the USB and energy monitoring components, which would be desirable when the TDB is in sleep. However, we have measured power consumption with these components removed at 18mW. Of this, 11mW are due to the 48MHz oscillator. This could be significantly reduced with a lower frequency oscillator, multiplied to 48MHz by using the FPGA's Phase-Locked Loop (PLL) module.

2.3.2 Application Setup

Our experiments use both TinyOS and Contiki applications without needing any extra programming effort since Aveksha is OS-agnostic by design. The only change to the OSes is the one required due to the change of the clock source that was required to synchronize the JTAG and application processor's clocks for PC polling (as shown in Figure **??**). The clock initialization module is responsible for creating the main processor clock from the hardware clock and for wiring other internal hardware clocks

to different sources. This module has to be changed in the OS for to work with our clock setup.

We use two TinyOS applications `TestNetworkLpl` and `TestFtsp` and an object tracking application in Contiki. `TestNetworkLpl` uses the collection tree protocol to push sensor readings to a base station [20]. The wakeup period is set to 128ms, which indicates how often the radio wakes up to test the channel for transmitting nodes. This is a typical application for sensor networks. We present a bug that was uncovered when we used the TDB to monitor all tasks in the watchpoint mode. `TestFtsp` is a time synchronization protocol [21]. Finally, LightTracker is a Contiki object tracking application takes light readings at each node and passes values that reach a threshold to the base station through a multi-hop routing protocol. The radio wakeup period is 125 ms [22].

2.3.3 Watchpoints

Using States to Monitor Energy: One application for the TDB is to monitor various state variables. Monitoring state variables and transitions is useful because they can be correlated to power consumption, and can aid in understanding the behavior of applications (as argued in Quanto as well [13]). This is particularly true in TinyOS, where the event-driven model encourages the use of explicit state machines.

In `TestNetworkLpl`, we have instrumented the application layer, low-power-listening layer, and the radio layer to monitor state changes. The instrumentation is simply to place a `nop` instruction which can be used as a trigger in the watchpoint mode. In the application layer, the beginning of every task and event handler is instrumented. In the low-power-listen layer, the state changes of interest are in the `RadioPowerState` module. This uses the state component interface in TinyOS, which we instrumented with `nop` instructions. In the radio layer, state variables have the

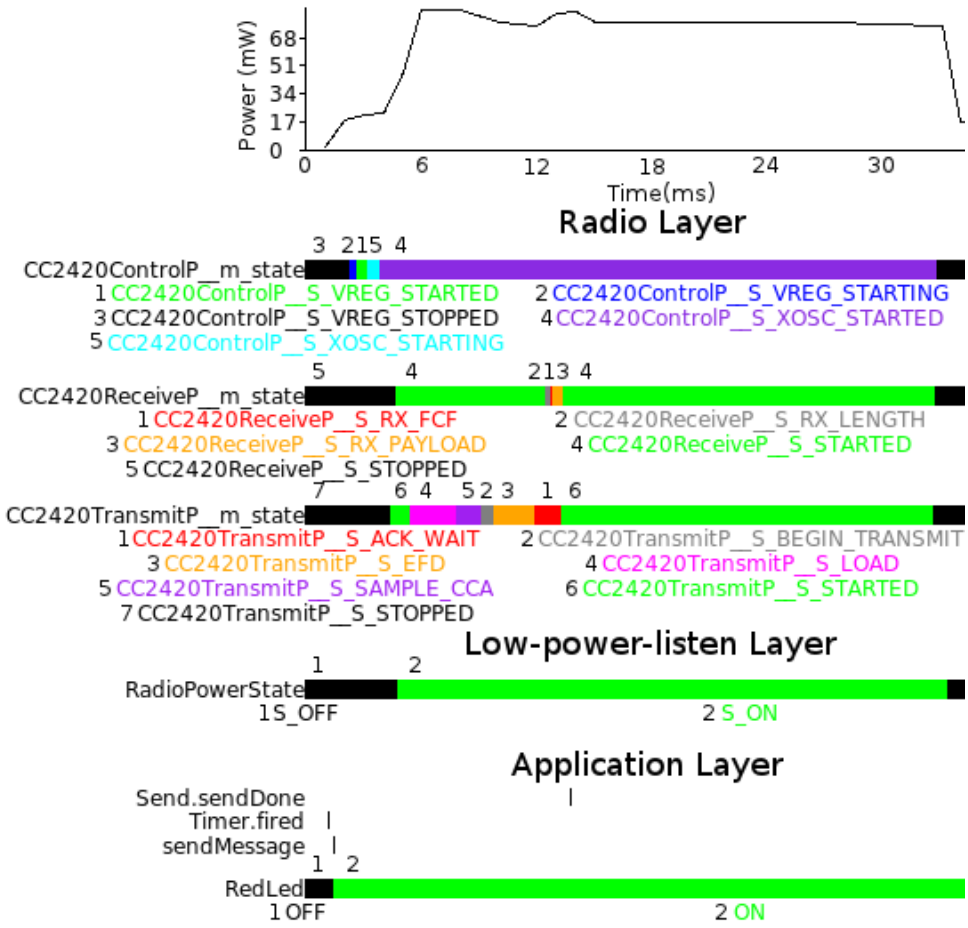


Fig. 2.7.: Watchpoint trace of states when sending a message in `TestNetworkLpl`, showing the application, low-power-listening, and radio layers. The number above each state's timeline corresponds to the numbering of the states under the timeline. For example, in the low-power listen layer, state 1 is `S_OFF` and 2 is `S_ON`; at the beginning the state is 1, then an extended period of state 2, followed by a return to state 1.

postfix `m_state`. We wrote a script that finds all assignments to these variables in the code and inserts a `nop` statement after the assignment.

Figure 2.7 shows a packet send that is initiated from the application layer when `Timer.fired` is triggered. The first step is to turn on the radio by starting the voltage regulator and oscillator, which is given by the state variable `CC2420ControlP_m_state`. The start up takes 1.6 ms, the duration of the `VREG_STARTING`, `VREG_STARTED`, and `XOSC_STARTING` states. The `CC2420TransmitP__m_state` shows the process of transmitting a message. The message transmission takes place during the states `S_BEGIN_TRANSMIT` and `S_EFD`. After the message is transmitted, the sender waits for an acknowledgment, which is shown in the `CC2420ReceiveP__m_state`. This variable shows the acknowledgment being received at 12 ms (the `S_RX_FCF` state) after which it is read off from the radio layer (the `S_RX_PAYLOAD` state). After this, the radio turns off at 32ms. A parameter of LPL controls the delay after receive and the default is set to 20ms which is verified by our experiment. This kind of low-level tracing of events in the stacks is useful for a developer wanting to get a detailed understanding of how a high-level function is accomplished (in this case, transmission of a message which requires an acknowledgment). Such an understanding can be used for performance tuning (speeding up some event in the time line, or reducing the amount of time spent in a particular state) or for energy optimization (knowing some energy-expensive state, reduce the amount of time the node spends in that state). This level of tracing would be very difficult to obtain through purely software means because of the fine-level of instrumentation that will be required, and correspondingly the high level of perturbation that will be caused to the normal execution of the application. On the other hand, Aveksha does not have to make tightly coupled changes to the hardware (the radio in this case), which are difficult to make and in some cases impossible when the hardware or the firmware is closed source.

Using Tasks to Debug an Application: The original objective of this experiment was to trace the collection tree protocol in the watchpoint mode. However, during the tracing, we observed some suspicious behavior that caused us to suspect that there was a bug in the low power listening layer of TinyOS. This was discovered by instrumenting

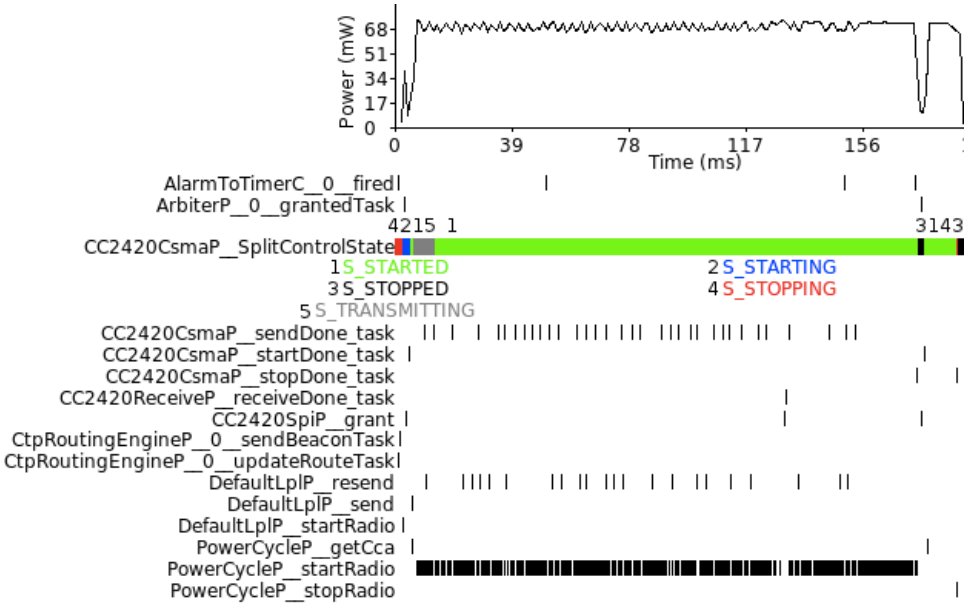


Fig. 2.8.: Watchpoint trace of task executions during a radio start event. The `PowerCycleP__startRadio` task is called over 3000 times due to a bug in the handling of the `CC2420CsmaP__SplitControlState`.

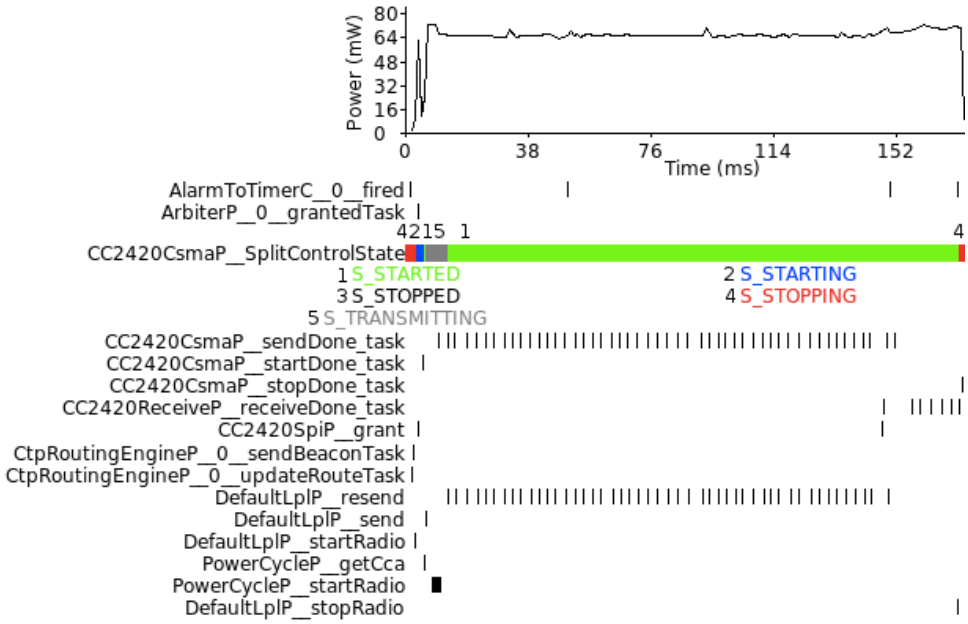


Fig. 2.9.: Watchpoint trace of task executions with the `startRadio` bug fixed.

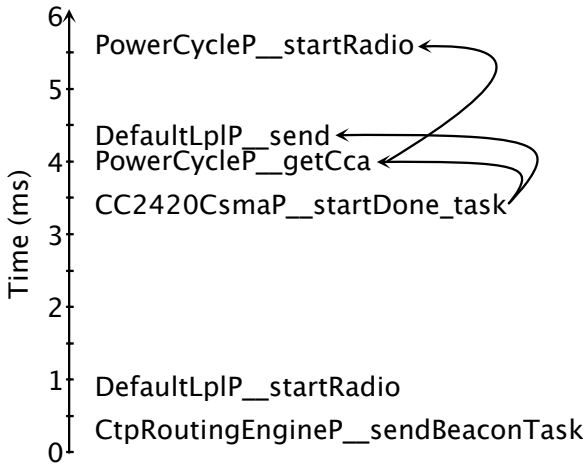


Fig. 2.10.: Execution timeline that causes task spinning.

all of the tasks in TinyOS for the `TestNetworkLPL` application. The nesC compiler in TinyOS creates a function called `SchedulerBasicP__TaskBasic__runTask` that contains a switch statement with a case for every task. By inserting a `nop` into each case, we can monitor every time any task is executed. We originally found that Aveksha was unable to keep pace with the rate of events that is generated after the mote is started up. Later, it turned out that this was due to a bug where some tasks were being repeatedly and unnecessarily re-posted.

Figure 2.8 shows a trace of the tasks shortly after the mote starts up. Three of the tasks (`PowerCycleP__startRadio, DefaultLplP__resend`, and `CC2420CsmaP__sendDone _task`) are stuck in a spin for 128ms. This implies that these tasks keep re-posting themselves and do not get any useful work done in each execution of the task. The spinning tasks are the result of the order of events that happen when the mote starts up. The timeline in Figure 2.10 shows the relevant events. First the collection tree protocol routing engine `CptRoutingEngineP` posts a task `sendBeaconTask` to send a beacon. This task results in the LPL module `DefaultLplP` posting a `startRadio` task. The radio is duly started after 3.4ms and the radio layer `CC2420CsmaP` sends a signal to `DefaultLplP` and `PowerCycleP` that the radio is started. `PowerCycleP` receives the signal first and

schedules the PowerCycleP__getCca task, after which DefaultLplP receives the signal and schedules the DefaultLplP__send task. Tasks are executed in the order they are scheduled in TinyOS. When PowerCycleP__getCca executes it schedules PowerCycleP__startRadio, which is executed after PowerCycle__send. The task PowerCycleP__startRadio fails, because the radio is already started. This is precisely where the bug lies. The radio has already been started and therefore this task should not re-post itself, but should return without doing anything. Eventually, when the sending of the beacon message has completed, the radio is set off to sleep and the PowerCycleP__startRadio task succeeds. This can be seen from the CC2420 state of S_STARTED toward the end of the timeline (when it is started a second time to perform the CCA).

The buggy version of PowerCycleP__startRadio is shown first.

```
static inline void PowerCycleP__startRadio__runTask(void) {
  if (PowerCycleP__SubControl__start() != SUCCESS) {
    PowerCycleP__startRadio__postTask();
  }
}
```

The undesirable effect of the bug is that fills a slot in the task queue (though a redesign in TinyOS 2.x limits this effect) and a task is being re-posted and invoked uselessly thus using up CPU resources.

The above is a real-case where the bug is activated. We hypothesize the following plausible application case where the bug will be activated and the PowerCycleP__startRadio task will *never succeed and will keep spinning endlessly.* Consider an application that starts sending a message and shortly afterwards (after the radio has finished S_STARTING and entered S_STARTED state) turns off low power listening. A low power listen interval of 0 indicates that low power listening should be shut off and the radio left on in receive mode. In this case, the task PowerCycleP__startRadio will never have the SUCCESS condition and will continue

to spin until the low power listen interval is again changed. We have confirmed that this happens when the following synthetic application is executed.

```
event void RadioControl.startDone(error_t err) {
  sendMessage();
  // Schedule the timer to fire while
  // PowerCycleP__startRadio is spinning
  call Time.startOneShot(100);
}
event void Timer.fired() {
  call LowPowerListening.setLocalWakeupInterval(0);
}
```

To fix this bug, consider what happens to the state of the CC2420 radio (shown as `CC2420CsmaP__SplitControlState` in Figure 2.8). The function `PowerCycleP__SubControl__start()` tries to start the radio and tests the state of CC2420. If the state is `STARTING` it returns `SUCCESS`, if the state is `STARTED` it returns `EALREADY`, and if the state is anything else it returns `EBUSY`. Therefore, the simple fix to the task `PowerCycleP__startRadio` is as follows.

```
static inline void PowerCycleP__startRadio__runTask(void) {
  if (PowerCycleP__SubControl__start() != SUCCESS
  && PowerCycleP__SubControl__start() != EALREADY) {
    PowerCycleP__startRadio__postTask();
  }
}
```

Figure 2.9 shows that this fix indeed stops `PowerCycleP__startRadio` from spinning (except for the short time the radio is in the transmitting state).

Processes in Contiki: An advantage of our approach over software tracing is that it is independent of the OS being used. Without any modification to the Contiki OS, the TDB is able to generate a trace of an application. In Figure 2.11, we show a

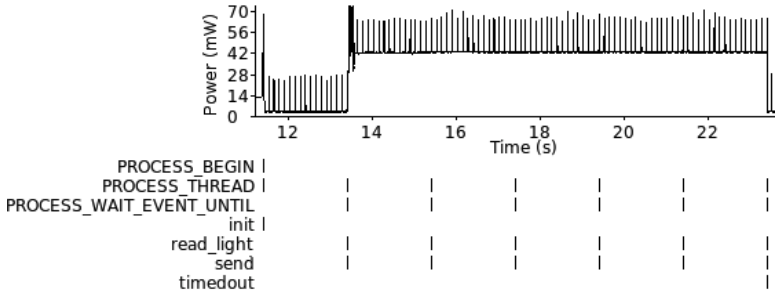


Fig. 2.11.: Watchpoint trace of application level functions and threads of a sender node in the Contiki tracking application.

trace of a sender node of a simple object tracking application called *LightTracker* [22], implemented in Contiki [23] version 2.4. LightTracker tracks a moving light source in a sensor network. There are two types of nodes present in the network: a *base station* and a set of *sender node*s. A sender node periodically (every 2 seconds) collects light intensity using its light sensor and forwards it to the base station, possibly in a multi-hop manner, if the sensed value is above a threshold. The base station periodically checks the received samples and selects the node with the maximum light intensity. The selected node is considered to be the current position of the light source.

Unlike TinyOS, Contiki features the use of threads as a key design component. This reduces the need of maintaining explicit state machines in the code. In the sender application, we place a `nop` at the starting point of every thread command. `PROCESS_BEGIN` represents the creation of a thread and is performed once at time 11 seconds. `PROCESS_THREAD` is executed every time the thread is started. `PROCESS_WAIT_EVENT_UNTIL` is a wait statement in the thread that blocks until 2 seconds have passed on the timer. The power spikes correspond well with the 125ms radio wakeup period.

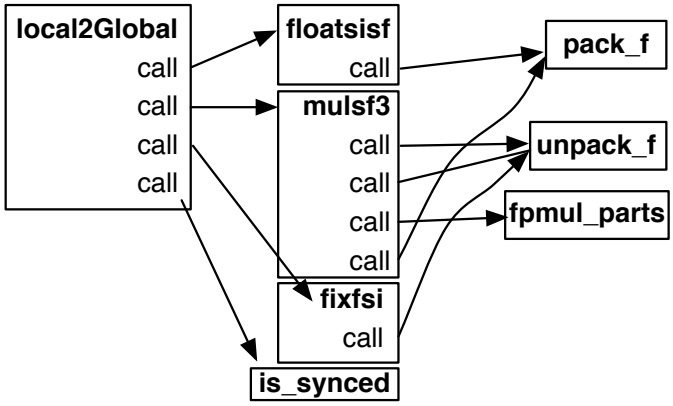


Fig. 2.12.: Call graph of the `local2Global` function in FTSP.

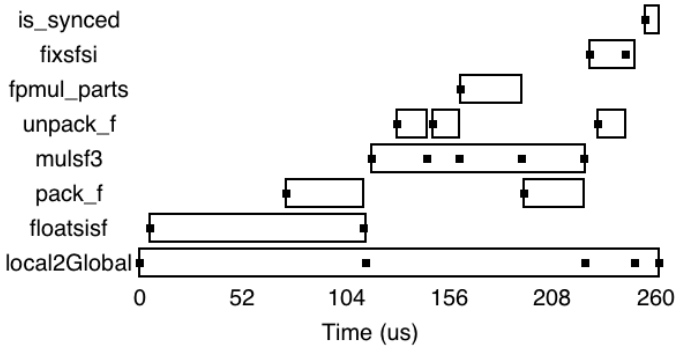


Fig. 2.13.: Trace of the functions invoked in one execution of the `local2Global` function. Filled squares represent data collected from PC polling, and open rectangles represent the inferred executions of each function.

2.3.4 PC Polling

Sampling the PC counter is a quick and non-intrusive operation. It does not have the flexibility of setting watchpoint triggers for specific conditions; however, it has the advantage of being able to measure events with greater timing accuracy than watchpoint polling. This is both because it is faster to take a sample of the PC counter and because it does not have to do buffer management.

A useful application of PC polling is for statistical profiling of an application, say to determine what parts of the code are most active. Raw PC polling data cannot

directly give a profile of the number of times a function is called or the total time the function takes to execute, inclusive of the times of the nested functions that it calls.

However, it is sometimes possible to combine knowledge of the call graph with the raw PC polling data to discover how many times a function is called and how long each instance executes. As an example, we examine the call graph of the `local2Global` time conversion function, in the TinyOS implementation of the synchronization protocol FSTP. This function requires a few calls to expensive 32-bit math functions. The order of calls in this function is fixed, and is shown in figure 2.12. In figure 2.13, the logged events generated from PC polling are shown as filled squares. PC polling uses a function lookup table, as described in Section 2.1. Every time a new function is entered an event is logged. Because the PC polling rate is equivalent to 6.4 application processor cycles, we are likely to capture every transition between functions. Combining the events with the call graph we can determine when functions start and end as shown by the open rectangles.

For this experiment, events are timestamped with a precision of 1 μs, which is less than the 1.6 μs a single PC poll takes. From the start of the execution of `local2Global`, to the call of `floatsisf`, the TDB recorded 5 μs (equivalent to 20 cycles of the application processor). A manual inspection of the assembly code reveals that the correct cycle count is 19, which is within the expected margin of error.

2.4 Related Work

There are primarily three areas of work related to Aveksha, namely power measurement, software and hardware for debugging sensor networks, and hardware support for debugging embedded systems.

Power measurement: The problem of estimating or measuring power (or energy) consumption has been addressed extensively in the context of various electronic systems. We restrict our discussion of prior work to techniques that specifically target sensor networks. Various sensor network simulators, such as PowerTOSSIM,

AVRORA, and COOJA provide energy estimation capability based on pre-built power models of the target hardware platform. Measuring (as opposed to estimating) the power consumed by a sensor node is usually done using the so-called *sense resistor* approach (see Section 2). SPOT [15] is an energy meter for wireless sensor nodes that is based on the sense-resistor approach and uses a voltage to frequency converter to transform the voltage samples into an energy counter that can be read by the sensor node. iCount [19] is an energy meter design that targets sensor nodes that have a switching regulator. It provides energy metering capabilities at almost zero cost by just counting the cycles of the switching regulator. The Energy Endoscope project [12] uses a separate application-specific integrated circuit (called EMAP2), implemented using a micro-power fuse-based FPGA, to perform charge accumulation based on the sense-resistor method. Similar to designs such as SPOT, Aveksha provides energy measurement capability with a large dynamic range.

Quanto [13] builds on iCount by using regression models to obtain per-component energy consumption based on the aggregate measurement provided by iCount and also performs energy accounting to various application tasks through causal activity tracking. Activities are tracked by calls to a software-based logger. Currently, Aveksha does not have the ability to assign energy consumption to individual activities, however, Aveksha could replace the Quanto software-based logger by using watchpoints to indicate activities. This would enable use of the Quanto algorithm to attribute energy consumption.

Sensor network debugging: Replay debugging is a well known technique for embedded systems [8] and has also been proposed for sensor networks [10]. Envirolog presents a software-only solution for recording events to flash memory [9]. Applications are annotated to indicate what should be recorded, which a preprocessor then turns into C code. During recording, 16 to 1024 bytes of RAM are used to buffer events which are then stored to Flash. FlashBox adopts a hybrid hardware/software approach to eliminate the bottleneck of writing to Flash [16]. In FlashBox, a second MCU and flash memory are added to provide dedicated recording. A recent software

solution [11] has focused on a specific kind of tracing (control flow tracing) and combines intelligent static analysis with run-time trace compression to decrease overhead. Nevertheless, this technique still requires applications to be instrumented to gather the tracing information. In contrast, Aveksha not only requires no modification to the application, but is also completely agnostic to the OS used. The above techniques have the disadvantage that they either perturb the timing behavior of the application, possibly suppressing some subtle bugs, or cause a large slowdown in application execution.

Emulators such as Avrora and Cooja provide the ability to customize tracing and profiling of applications. In Avrora an interface is provided for writing custom plugins to monitor the emulation. YETI extends Cooja with a GDB proxy [24]. This allows using the GDB debugging tool to set breakpoints and watchpoints in the code being emulated. Similar extensibility could be added to Aveksha by providing an API to customize three points: the watchpoint conditions set by JTAG, the event filtering algorithm performed by the FPGA, and how the MCU stores the data.

Hardware support for debugging embedded systems: Real-time trace functionality has been implemented in many processor architectures. For example, the CoreSight Trace Macrocells provides hardware cores that can be added on as peripherals to an ARM-based system-on-chip to produce a cycle-accurate trace of execution. This includes the ability to collect and compress a large amount of trace data on chip and to transfer this data to a trace port interface unit, such as JTAG. The MSP430 MCU used in the Telos mote has a limited built-in OCDM. It is typically used by tools such as IAR [17] to record the last 8 instructions executed before a watchpoint or breakpoint. Aveksha goes beyond this by demonstrating that it is possible to provide CoreSight like functionality on a low end MCU commonly used in wireless sensor nodes. Hardware designed to interface an OCDM to a host computer via the JTAG standard is often referred to as an In-Circuit Emulator (ICE) or In-Circuit Debugger (ICD), or more correctly, a JTAG adapter. Many ICE tools are available for the MSP430 processor family. An example, that inspired the authors to under-

take this work, is the open source GoodFET [25]. However, the GoodFET is only capable of operations involving reading and writing to flash. The debugging features of the OCDM (e.g., breakpoints, watchpoints, state storage) are not documented by Texas Instruments, and the protocol to activate these features was reverse-engineered. There exist several point solutions for hardware meant for tracking different kinds of control flow for the purpose of debugging, e.g., in [26], the authors design a hardware ASIC that monitors loops taken by tasks in a multi-tasking environment and performs this in a non-intrusive manner to the application.

3. SOFTWARE BASED SYSTEM-LEVEL RECORD AND REPLAY OF WIRELESS SENSOR NETWORKS

3.1 Introduction

In this Chapter we present a sofware based approach to system-level record and replay of WSNs.

3.1.1 Challenges of Record and Replay in WSNs

Realizing record and replay for WSNs (and to some extent in other embedded systems) is challenging for several reasons:

Resource constraints: The record system must fit within the bounds of the severe resource constraints typical of WSNs. In an effort to reduce the cost, size, and energy consumption of sensor nodes, the main processor and non-volatile storage are heavily constrained in WSNs. The main processor is typically a microcontroller (μC) which may be limited to a few MHz and RAM in the range of tens of KBs. Non-volatile storage is usually a flash chip which may contain anywhere from a MB to a few GB of storage. As two points of reference, the TelosB sensor node has 1 MB of flash and the top-end Shimmer sensor node has a 2 GB SD card for storage. Compared to the volume of raw trace data generated during record, this storage capacity is tiny. For example, we have observed traces of generated at 1 MB per minute in an experiment detailed in Section 3.3. Additionally, storing data to flash is energy expensive, with frequent flash usage reducing a node's lifetime by a factor of 3 [27].

Real-time constraints: WSNs are cyber-physical systems with soft real-time constraints. Adding instrumentation to record non-deterministic events can interfere with the timing of the application and cause it to miss its deadlines.

Portability: There are many operating systems (OSes) for WSNs, the two most popular being TinyOS [28] and Contiki [23]. Also it is not uncommon in embedded system development to run directly on the "bare metal" with no OS support, as demonstrated by the GoodFET JTAG adapter [25]. Manual modification of OS drivers or system libraries, as done in previous record and replay systems [29, 30], hinders adoption. For this reason we seek a solution which requires minimal OS specific adaptations. The solution should take the source code of the firmware to be installed on the node, and produce an instrumented version which can run directly on "bare metal".

System-level replay: WSNs often do not have hardware enforced separation between application and system software, due to a lack of hardware support on many μCs, and prominent WSN OSes such as TinyOS [28] and Contiki [23] do not have a clean separation between system and application code. This calls for solutions that record the complete execution of a sensor node's processor for replay, rather than only application components as done in work such as `liblog` [29]. We call this *system-level* record and replay, which is more expansive in scope than application-level record and replay.

3.1.2 Tardis Approach

In this Chapter, we present the design and development of a software-only system-level record and replay solution for WSNs called Tardis. In short, we address the four challenges described above by handling *all* of the sources of non-determinism and compressing each one in a resource efficient manner using respective domain-specific knowledge. For example, one type of non-determinism is a read from what we call a *peripheral register*. These are registers present on the μC chip, but whose

content is controlled from sources external to the main processor. Reads to a register containing the value of an on-chip analog-to-digital (ADC) converter are sources of non-determinism. We can reduce the number of bits that must be stored for tracing them by observing that an ADC configured for 10-bit resolution in fact only has 10 bits of non-determinism, despite the register being 16 bits in size.

The compression scheme for each source of non-deter- minism is informed by a careful observation of the kinds of events that typically occur in WSN applications, for example, the use of register masking which reduces the number of bits which must be recorded—instead of the full length of the register, only the bits that are left unmasked need be recorded. The compression schemes are also chosen to be lightweight in their use of compute resources. Furthermore, the compression is done in an opportunistic manner, whenever there is "slack time" on the embedded μC so that the application's timing requirement is not violated. By using the different compression schemes in an integrated manner in one system, *we are the first to provide a general-purpose software-only record and replay functionality for WSNs*. By "general-purpose" we mean that it can record and replay *all* sources of non-determinism and thus Tardis can be used for debugging *all* kinds of bugs, whether related to data flow or control flow. Previous work in software-based record and replay for WSNs has captured only control flow (e.g., TinyTracer [11]) or only a predetermined subset of variables and events (e.g., EnviroLog [9]).

3.1.3 Contributions

This work makes the following four contributions through the design and development of Tardis.

1. We make seven domain-specific observations about the events that are the sources of non-determinism in a WSN. These observations lead us to specific ways of compressing each respective kind of event.

2. We describe the first general-purpose software-only re- cord and replay solution for WSNs. Our solution is general-purpose in that we record all sources of non-determinism at the system level, which also makes our implementation easily portable to other embedded OSes. We demonstrate this by collecting results from both TinyOS and Contiki.

3. We show with experiments on real hardware, that with the constrained resources of a typical WSN platform, we generate a 53-79% smaller trace size compared to the state-of-the-art control flow record and replay technique [11], while being able to capture far more sources of non-determinism and thus able to replay an execution more faithfully.

4. We give the case study of diagnosing a previously unreported bug which had been in the TinyOS codebase for over 7 years. This bug is in the widely used Collection Tree Protocol (CTP), which is used for collecting data at a base station from multiple sensor nodes, in a multi-hop manner.

Tardis is available for download at
http://github.com/mtancret/recordreplay.

3.2 Design

This section introduces the design and implementation of Tardis.

3.2.1 Overview

The main capability of Tardis is to replay in an emulator the original run of a sensor node faithfully down to each instruction and the sequence between instructions. Deterministic replay is achieved by starting from a checkpoint of the processor's state, and then replaying all sources of non-determinism [30]. There are two broad sources of non-determinism in WSNs: external inputs from memory mapped I/O and the type and timing of interrupts. We will use the term *peripheral registers* to refer to

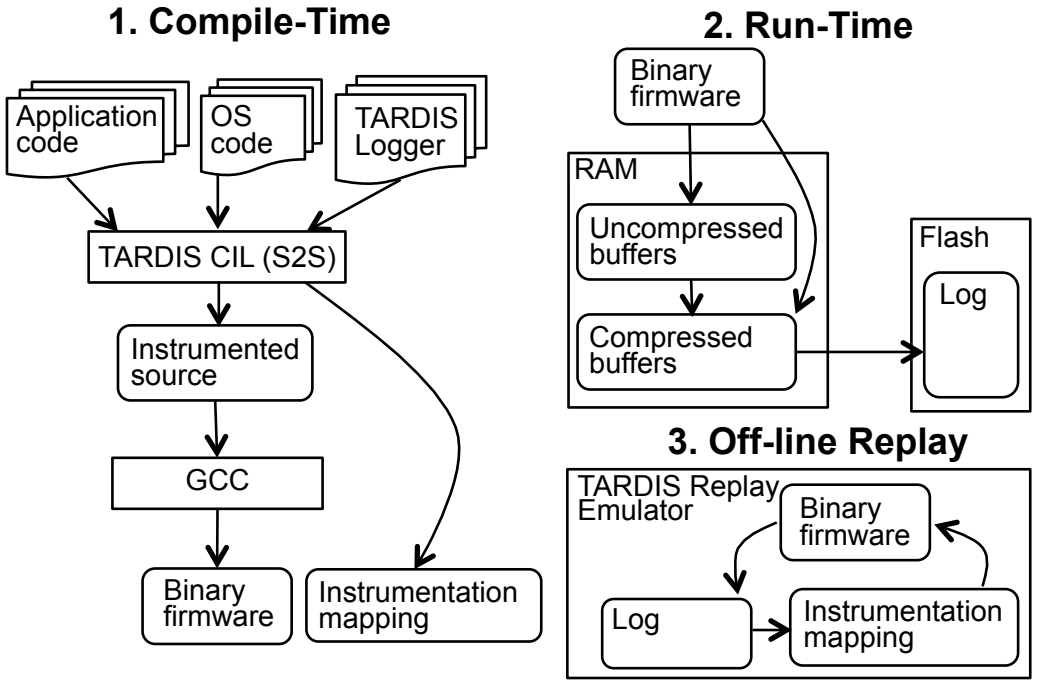


Fig. 3.1.: The TARDIS debugging process consists of instrumentation at compile-time, *in situ* logging of trace data at run-time, and off-line replay during debugging.

memory mapped I/O, which includes registers that report the value of serial I/O, real-time clocks, interrupt flags, analog-to-digital converters, etc.

TARDIS is designed to be used *in situ* to record events in deployed sensor nodes for subsequent troubleshooting. The overall operational flow is depicted in Figure 3.1, which depicts three phases: compile-time, run-time, and off-line replay. In the first phase, a source-to-source C code compiler is used to insert instrumentation for recording. In the second phase, the node executes *in situ*, and logs a checkpoint and a trace of its execution to flash. It operates in the manner of a black box recorder; when the flash is full, a new checkpoint is taken and the oldest data is overwritten first. The third phase is the replay, which happens in the laboratory running an emulator on a (comparatively) resource-rich desktop-class machine. During execution of the application on the emulator, the trace of non-deterministic data is used to determin-

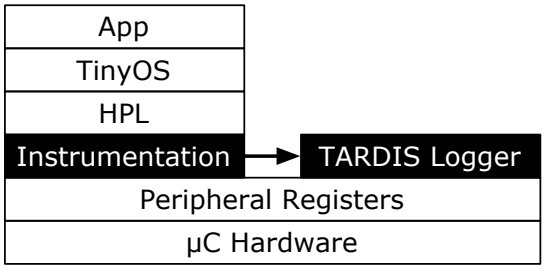


Fig. 3.2.: TARDIS instrumentation and logger with respect to the TinyOS stack.

istically reproduce the node's execution. These three phases as well as their basic implementations are discussed in the following sections.

3.2.2 Compile Time

Recording Peripheral Register Reads

One goal of TARDIS is to be able to record the reads of non-deterministic peripheral registers using as little OS-specific code as possible. This is achieved through an automated compile-time source-to-source transformation of the code that is to run on the sensor node. All instructions which read from peripheral registers are identified and instrumented, such that, the value of the reads are intercepted and passed to a logger during recording. This step assumes that a configuration file has been created to specify the non-deterministic registers of the target architecture.

We define *target code* as all code intended to run on the sensor (i.e., OS and application) — WSN OSes are typically monolithic in that all target code is compiled together into a single binary firmware. Figure 3.1 presents the compile-time process of producing an instrumented binary. First, code files from the application, OS, and TARDIS logger are fed into TARDIS CIL, a source-to-source transformer based on the C intermediate language (CIL) [31]. TARDIS CIL identifies instructions that read from peripheral registers and instruments them, producing an instrumented source file as one of its outputs. The other output from TARDIS CIL is an instrumentation

mapping file which gives the location of each instrumented instruction and the type of encoding or compression applied to the logged value. A compiler, in this case GCC, then compiles the instrumented source into a single binary to be installed as firmware on the sensor node. In the case of TinyOS, the instrumentation sits between the peripheral registers and the hardware presentation layer (HPL), as shown in Figure 3.2. The HPL is the layer of code thorough which all access to external I/O must pass. However, because TARDIS identifies register reads in C code, it does not require the target code to have an explicitly defined HPL, for example, we tested TARDIS during its development on C code written for "bare metal" (i.e., without an OS). At runtime, after intercepting the value read from a peripheral register, the instrumentation passes the value to the TARDIS logger.

The replay part of TARDIS uses the instrumentation mapping file to decide which instructions access peripheral registers and thus need to be fed from the log, and then to determine how to decode the items in that log.

One alternative design approach would have been to manually instrument the HPL to intercept all reads from peripheral registers. Such an approach would be similar to `liblog`, in which a shared library (`liblog`) was created to intercept calls to `libc` [29]. Not only would this approach add to the manual effort of porting TARDIS to different OSes, but it would also miss out on opportunities to reduce log size by not considering the context in which a peripheral register value is being used in the code. For example, we describe in Sections 3.3.4 and 3.3.3 how the identification of masking of peripheral register values and polling loops can help to reduce or completely eliminate logging requirements.

Recording Interrupts

To replay interrupts, TARDIS logs the instructions at which interrupts are delivered during run-time, and redelivers the interrupts at the same instructions during replay. The instruction can be uniquely identified by the combination of the interrupt's return

address and the loop count. TARDIS instruments every loop body in the target code with an increment instruction on a global counter variable — *the loop counter*. An alternative approach is to use a hardware based performance counter to count the number of branches, however, such counters are not often found in μCs [30].

3.2.3 Runtime System

A single binary containing the application, OS, and TARDIS code is programmed into a sensor node. The runtime code consists of buffering, encoding, compressing, and storing logs to flash. The reads to peripheral registers are intercepted by TARDIS's instrumentation and it then passes it on to the logger. Note that this assumes that peripheral registers are accessed directly, rather than through indirect addressing; we explore the implications of this assumption further in Section 3.5. The update function performs the check for buffers that are ready to be compressed or written to flash. Compression and writing to flash happen *asynchronously* from the calls to log values so that they do not interfere with the real-time execution of the application. The scheduling of invocation of the update function is OS specific — in TinyOS it is called after every task execution and interrupt, while in Contiki it is called after every thread switch and interrupt. Also TARDIS must share the flash with the OS requiring resource arbitration code.

The code for TARDIS is mostly OS agnostic. There is a small amount of code specific to the OS being instrumented, it includes calling the TARDIS initialization and update functions. For example we added 23 lines to TinyOS and 39 lines to Contiki.

3.2.4 Replay

Replay is performed centrally, say at a lab computer, rather than at the nodes; this is similar to the design of the overwhelming majority of record-and-replay solutions, in embedded domains and otherwise. This means the checkpoint and log must be

collected at a central location. An emulator (mspsim in our case [2]) is modified to deliver non-deterministic register values and interrupts to the application during replay. The emulator starts from a memory checkpoint or known starting state (e.g., boot-up). For replay, the binary is executed until a register read or the next interrupt in the log is encountered. Whenever a read from a peripheral module register is encountered, a map file generated at compile time is consulted to determine how the register has been encoded. Based on this information the register is decoded from the log. The emulator also knows the next interrupt in the log. When the return address and loop count match the next interrupt in the log, the interrupt is executed. Since all sources of non-determinism recorded during runtime are fed into the emulator, this faithfully reproduces the execution.

3.2.5 Debugging Workflow

A typical workflow for debugging in Tardis starts with simple invariants used to check for the correct operation of the network. For example, an invariant at the basestation may check that no more than a threshold amount of time has passed since the last message was received from each node in the network. When any invariant is violated the basestation broadcasts a command to all nodes in the network to not overwrite their current traces. The broadcast is performed using a common dissemination protocol, which does not depend on the routing protocol. Then a programmer is alerted of the problem. The programmer can wirelessly collect traces from the nodes in order to replay their execution.

3.3 Encoding and Compression of Non-Deterministic Data

This section describes how we efficiently trace non-determinism in Tardis.

Table 3.1.: Summary of key ideas and benefits of Tardis compression methods.

Observation	**Design**	**Result**
Some registers are deterministic	Consult table of register definitions	26.8% log reduction
Can skip polling loops during replay	Detect and ignore polling loops	25.9% log reduction
Register reads are often masked	Detect and ignore masked bits	56.7% log reduction
Nodes spend most time in sleep	Return address is predictable when interrupt during sleep	12.7% interrupt log reduction
Small delta between reads of timer	Use delta encoding	72.7% timer log reduction
State registers are highly repetitive	Run length enconding	47.8% state log reduction
Data registers compressible with general algorithms	LZRW-T	65.7% data log reduction

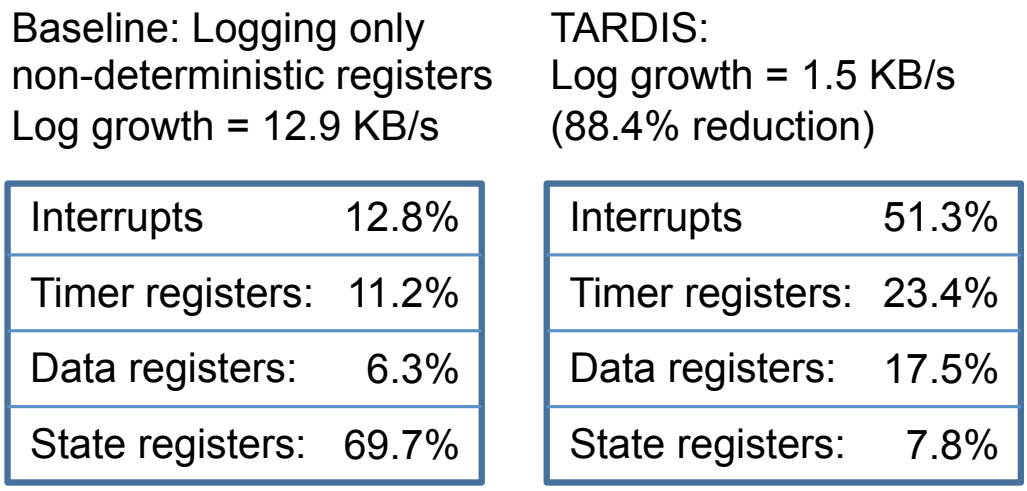


Fig. 3.3.: Comparison between baseline and TARDIS.

3.3.1 Overview

Up until now we have described the non-deterministic data required for replay and how TARDIS instruments the target code for logging. One may wonder why a simple design does not suffice — record all the sources of non-determinism during execution on the node and store them to stable store, then bring the trace back to a central node for replay. This is due to the fact that the rate of non-deterministic data is too high for the resources of today's WSNs, even for simple regular applications.

For example, consider the TinyOS application MultihopOscilloscope (MHO) that collects sensor data at each node at a one second interval and propagates the data to a base station at a five second interval. We ran MHO on a small five-node network with all nodes in radio range of the base station and Low Power Listening (LPL) enabled with a wake- up interval of 64 ms. (More details about this experimental setup can be found in Section 3.4.1.) For this configuration, we found that recording all interrupts and reads from peripheral registers produces a log at a rate of 15.1 KB/sec. Ignoring deterministic registers (e.g., peripheral control registers), the log rate is 12.9 KB/sec. In this paper, we use logging all interrupts and only the non-deterministic registers as our baseline, as described in Section 3.3.2. A rate of 12.9 KB/sec would fill the 1 MB

flash on the TelosB in 78 seconds. The flash shares the I2C bus with the radio, and with a write throughput of 170 KB/sec, the flash would require a 7.6% utilization of the I2C bus. This could interfere with the application. Additionally, it increases the average power consumption by 4.4 mW, which is significant to the TelosB which has a sleep current of just 3 μA.

In the following we describe how we meet the challenges through careful selection of what to record (Sections 3.3.2-3.3.5) and encoding and compression (Sections 3.3.6-3.3.8). Using observations of typical WSN applications and deployments to guide our design, we are able to achieve significant compression using low cost compression techniques. We structure the description of each technique as the observation we glean from many WSN applications and hardware architectures, followed by the technique we implement in TARDIS, and then giving the quantitative result to show the effectiveness of the technique (Table 3.1 presents an overview of these findings). These results are collected from the TinyOS application MHO running on actual TelosB motes in a network configuration described above. The snapshot of the results are shown in Figure 3.3. The evaluation section shows the overall benefit of TARDIS, with all these techniques operating together, for a wider variety of applications.

3.3.2 Non-determinism of Registers

Observation: It is not the case that all of the peripheral registers are non-deterministic. Some of the peripheral registers are used for configuring the peripheral. For example, IE1 is an interrupt enable register, which is used to enable particular interrupts. The value of this register is only set by software and is therefore deterministic. Even for those registers which are non-deterministic, it is sometimes the case that not all of the bits in the register are non-deterministic. For example, ADC12CTL1 is a 16-bit register that is used by software to control the ADC. However, it has a single non-deterministic bit, which acts as a flag to indicate whether the ADC is busy.

Design: TARDIS avoids recording reads from deterministic registers by consulting a register mapping file that specifies which registers and which specific bits are actually non-deterministic. This file must be manually created once for each processor architecture.

Result: Logging only non-deterministic registers as opposed to all peripheral registers results in a 14.5% reduction. For all remaining results, logging only "non-deterministic registers" is the baseline. Logging only non-deterministic bits is a 14.4% reduction over baseline.

3.3.3 Polling loops

Observation: Polling loops are commonly found in embedded systems code. An example of a polling loop is where the μC transmits a byte to the SPI bus for network communication, then it stays in a loop until the transmit complete flag is set, before transmitting the next byte. The following code is taken from Contiki where `IFG` is the interrupt flag register and `TXFLG=1` is a mask for the least significant bit that is cleared when transmission is finished for this byte.

```
while (IFG & TXFLG);
```

Design: In the example, `IFG` is read multiple times before the byte has finished being transmitted. Normally, TARDIS would log each read. However, the loop itself does not modify global or local memory, and it will eventually exit. Therefore, it is safe for replay to simply skip beyond the loop without losing the property of deterministic replay. There is however one consequence of skipping the loop, and that is losing the cycle accuracy of the replay. However, the time to transmit a byte is predictable, particularly because the SPI bus uses a multiple of the main CPU clock for timing, and can be accounted for by the replay emulator without needing to keep track of how many times the check executed.

Result: Removing polling loops reduces the log size by 25.9% relative the baseline.

3.3.4 Register Masking Pattern

Observation: Register masking is a common pattern in embedded systems. Take for example the case of the interrupt flag register, where each bit represents a different condition. It is common to test one specific condition, so a register value is bitwise ANDed with a mask, which typically has a single bit set as one. In the following line of code, a mask (`TXFLG`) is applied to test if the transmit flag is set in the interrupt flag register (`IFG`).

```
not_done_transmitting = IFG & TXFLG;
```

Design: It is sufficient to record only the value of the unmasked bits. This can lead to a significant savings, in the above example a 16-bit register read can be stored as a single bit. Tardis CIL checks for the register reads that are masked, and instruments the recording of only the unmasked bits.

Result: Removing masked registers results in a reduction of 56.7% relative to baseline.

3.3.5 Sleep-wake Cycling and Interrupts

Observation: An important design feature of WSN programs is sleep-wake cycling — the ability of the node to spend most of its time in a low-power sleep mode where the main CPU clock is disabled and only wakes up for short bursts of activity before going back to sleep [32–34]. Without this feature the batteries of a mote such as the Telos would drain in days, rather than last for as long as 3 years at a 1% duty cycle.

Design: One of the sources of non-determinism is the timing of interrupts, which as described in Section 3.2.2 requires logging the interrupt vector (4-bits), the return address (16-bits), and the loop count (16-bits). This results in a 36-bit log entry for each interrupt. However, the entry can be significantly reduced for interrupts which wake the node from sleep. Upon reaching a sleep instruction, Tardis Replay knows that the next interrupt vector must be delivered, so the return address and loop count

do not need to be recorded. This is because there is only one way to exit sleep — an interrupt.

Result: Interrupt compression results in a 12.7% reduction of the interrupt log. The logging of interrupts accounts for only 12.8% of the baseline log, but they are 51.3% of the log after all other compressions have been applied.

3.3.6 Timer Registers

Observation: Timer registers are counters that are incremented on the edges of either a real-time clock or the main CPU clock. When a timer register either overflows or reaches the value of a capture/compare register, a timer interrupt is fired. The first read of a timer following a timer interrupt is likely to result in a value which is close to the value of the capture/compare register that caused the interrupt, or zero if the interrupt was caused by an overflow. The difference in time, and therefore value, between successive reads of timer registers is likely to be small. This is because all of the activity following an interrupt happens within a small period of time, due to low duty cycle operation, as pointed out in Section 3.3.5.

Design: The delta between subsequent timer reads is encoded. The exception is that first timer read following a timer interrupt is encoded based on the difference between the capture/compare register (or zero for overflow). TARDIS encodes the difference between the predicted value and the actual value using prefix codes to shorten the length of small values.

Result: Compression reduces the size of the timer log by 72.7%.

3.3.7 State Registers

Observation: Some of the peripheral module registers exhibit strong temporal locality. For example, a flag bit that indicates whether an overflow has occurred in a timer will usually be set to zero, because the overflow case is less common — when the timer expires. As another example, six capture/compare registers are used to

time different events. In a typical application, one of the capture/compare registers may be used to time a high frequency activity such as sampling a sensor, while the others are associated with less frequent activities. The registers with less frequent activity will contain the same state over long periods of time. The observation is that most reads to registers reporting the status of some peripheral module have the same value on consecutive reads.

Design: Because of the high level of repetition, state registers are compressed with Run Length Encoding (RLE).

Result: RLE reduces the state register log by 47.8%. In the baseline, state registers account of 69.7% of the log, but after all compressions have been applied they account for only 7.8% of the log.

3.3.8 Data Registers

Observation: Two common WSN features, sensors and radios, account for another source of non-deterministic reads. We classify the peripheral registers that contain sensor and radio data as data registers. These registers are quite compressible due to repeated sequences. For example, radio messages contain header information that is often similar from one message to the next. Sensor readings often have repeating values or slowly changing values because of the slowly changing nature of the physical environment.

Design: *Data* registers are compressed using a generic compression algorithm. We created LZRW-T, which is similar to LZRW [35], but implemented for the MSP430 processor. Like other LZ77 [36] based algorithms, LZRW-T uses the previously compressed block as a dictionary. The advantage of LZRW-T is that it has a very small memory footprint, which is critical in systems like WSNs that have only kilobytes of memory. LZRW-T is configured to use a sliding window of 128 bytes along with a hash table of 64 bytes for a total implementation of 192 bytes in RAM.

Result LZRW-T results in the reduction of logged data registers by 65.7%.

State/Timer Stream:

```
if type == state then write
   0b111<6-bit index><8-bit run_length><X-bit value>
if type == timer and delta < 4 then write 0b0<2-bit delta>
if type == timer and delta < 64 then write 0b10<6-bit delta>
if type == timer and delta >= 64 then write 0b110<16-bit delta>
```

Generic Stream (LZRW-T):

```
if no matching sequence found then 0b0<8-bit value>
if matching sequence found then 0b1<8-bit offset><8-bit length>
```

Interrupt Stream:

```
if loop_count == 0 then write 0b0<4-bit vector>
if loop_count < 256 then write
   0b10<4-bit vector><16-bit return_address><8-bit loop_count>
if loop_count >= 256 then write
   0b11<4-bit vector><16-bit return_address><16-bit loop_count>
```

Fig. 3.4.: Logging format.

3.3.9 Log Format

The log is stored in three independent streams: state/timer, generic, and interrupts. The bit format for the three streams is provided in Figure 3.4. Every read from a state register in the program's code is given a unique id. Indexing based on read instructions rather than register addresses is necessary, because two different read instructions of the same register may have a different number of non-deterministic bits that need to be stored due to masking as described in Section 3.3.4. Reads from timer registers are reproduced during replay in the same order as they were logged, so no index is required. The timer delta is stored as described in Section 3.3.6. Generic register reads are stored using the LZRW-T compression format. Interrupts require storing a vector, a return address, and a loop count. Loop counts are typically very small because the count is reset after every sleep period, which lends itself to compression using prefix codes. A loop count of zero eliminates the need to store a return address because the replay code knows that the next interrupt occurs immediately following the next sleep period.

3.4 Evaluation

In this section, we substantiate our claim that domain-specific compression techniques used by TARDIS can significantly reduce trace size yet operate with tolerable overheads. To evaluate TARDIS, we measured both the cost and the benefit for typical WSN applications from two widely used OSes (TinyOS and Contiki) running on real hardware (TelosB motes). Benefit is measured by the reduction in the size of the log stored to flash. Cost is measured by both static and runtime overheads. Runtime overhead is measured in energy and CPU usage, whereas the static overhead is measured in program binary size and RAM usage. As we show in the following, the domain-specific compression techniques used by TARDIS can significantly reduce the trace size – a 77%-92% reduction – whilst imposing only tolerable overheads. In addition, we demonstrate how trace sizes are reduced by 53-79% with respect to the state-of-the-art control flow record and replay technique TinyTracer [11] whilst enabling diagnosis of a much wider class of bugs than TinyTracer. Finally, we give the case study of diagnosing a previously unreported bug.

3.4.1 Experimental Setup

The experiments are conducted either with a single TelosB node, or in a network of 9 TelosB nodes. The 9 nodes are arranged in a grid with 1m separation between adjacent nodes, and the base station is at a corner. All of the nodes are in radio range of each other, which represents the worst case in terms of the rate of non-deterministic inputs, because of the radio traffic overheard at each node. The experiments involving a single node represent an inactive network (i.e., no radio traffic).

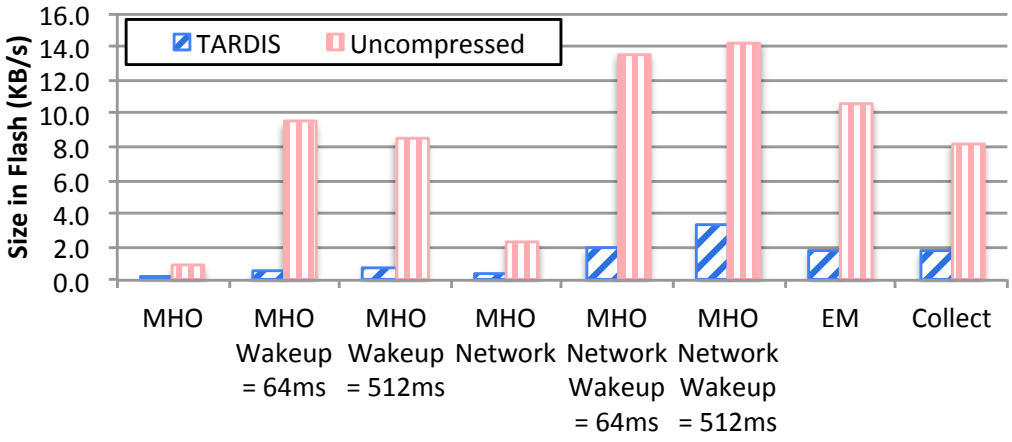
The experiments are run for three benchmarks. We chose two of the benchmarks, namely MultihopOscilloscope (MHO) and Collect, because they are representative middleware-type applications in TinyOS and Contiki respectively. We chose Earthquake Monitor (EM) as the third benchmark to test a higher-level application and

one that significantly stresses the sensors and the related computation to deal with the sensed data.

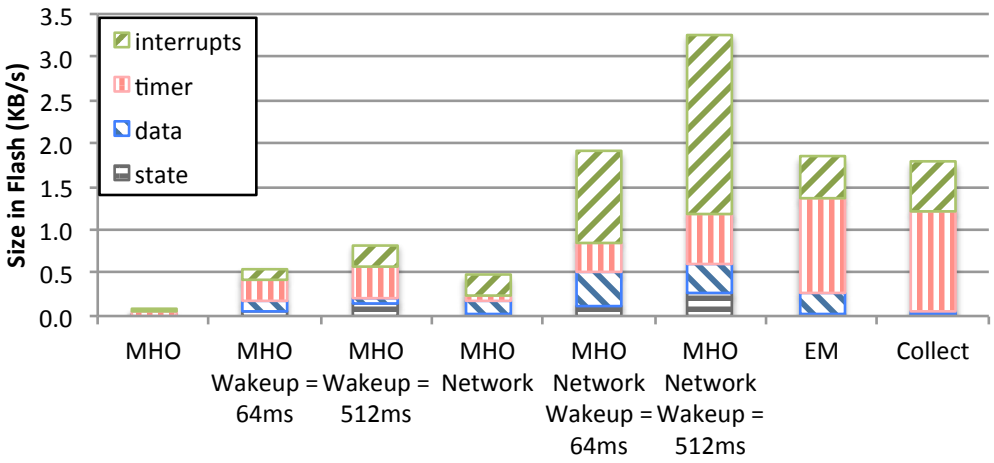
MultihopOscilloscope (MHO) is a typical data collecting WSN application. In MHO, each node samples a light sensor at a rate of 1 Hz, and forwards the measurements to a base station every 5 readings. By default MHO has the radio turned on all of the time. Energy savings come through enabling Low Power Listening (LPL), a Media Access Control (MAC) level protocol where the radio is turned on at a fixed interval to perform a Clear Channel Assessment (CCA), and immediately turned back off if there is no activity. In the results "MHO" indicates the node being recorded is alone and not in a network, "MHO Wakeup=" indicates LPL is employed with the given wakeup interval, and "MHO Network" indicates the node being recorded is in a network of 9 nodes. A higher wakeup interval means the node wakes up less often, and therefore is expected to generate less non-deterministic data. It is important to observe TARDIS behavior in a inactive network because that is the common condition in WSNs where LPL was designed to provide the most significant energy savings, as the radio is turned off most of the time.

Collect is an example application distributed with the Contiki operating system. Its purpose is similar to that of MHO; every 20 seconds each node sends a message containing readings for 5 different sensor sources.

Earthquake Monitor (EM) is a TinyOS application patterned after [37], however we reimplemented it for our experiments since the application was not available from the authors. In EM, each node samples an accelerometer at a rate of 100 Hz for 1 second. At the end of the sampling period it performs a Fast Fourier Transform on the sampled data, sends a message over the radio, and then begins the next sampling period. The sample rate of 100 Hz is considered sufficient for the application of earthquake monitoring [38]. We run EM in a single hop mode with each node sending directly to a base node.



(a) Rate of log growth for TARDIS and uncompressed.



(b) Size of different log components for TARDIS

Fig. 3.5.: Rate of log growth and the size of different log components for TARDIS. Uncompressed log rate shown for comparison.

3.4.2 Runtime Overhead

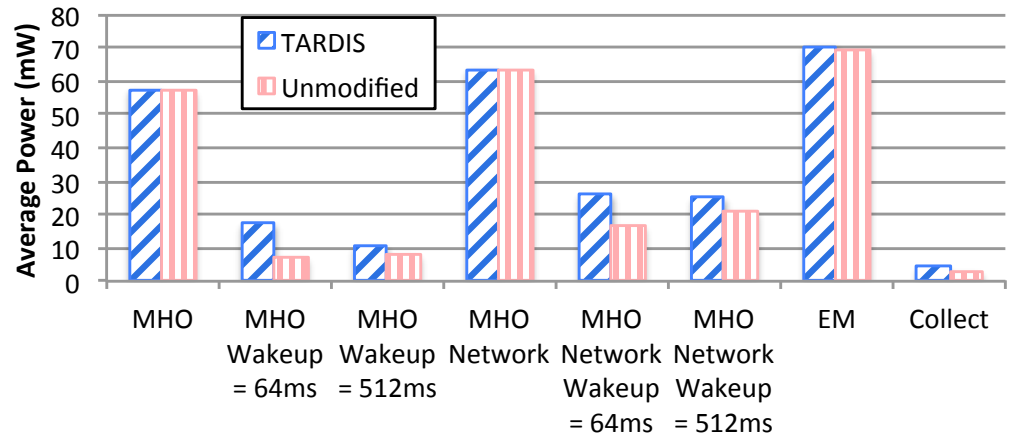
The runtime overhead includes the amount of flash used to store the log, and the additional energy and CPU time expended for tracing, a cost of using TARDIS.

Flash Size

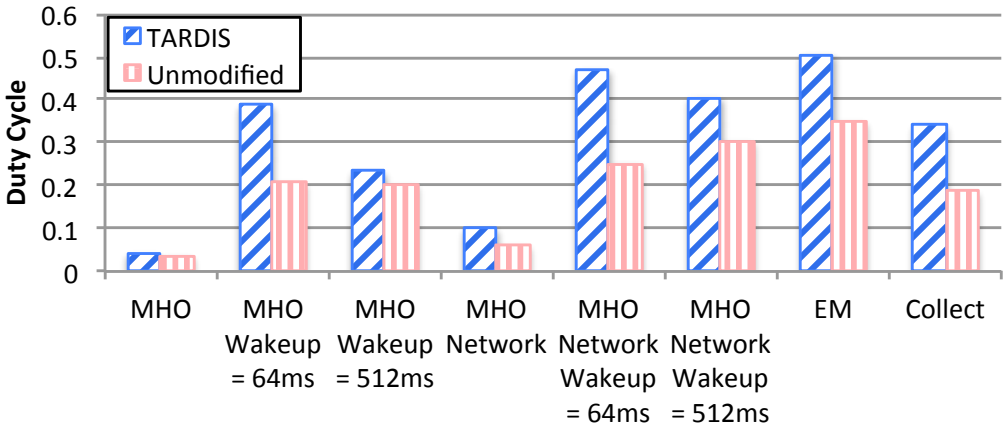
Figure 3.5(a) shows the rate of log growth for TARDIS and for an uncompressed trace. We see that for MHO single node, the log size is reduced by 92%, 94%, and 90% compared to the uncompressed traces for the various wakeup intervals—no LPL, 64 ms, and 512 ms respectively. For MHO Network mode, the reduction in log size is 80%, 86%, and 77% respectively. This points to the fact that with a lightly loaded network, there is both less source of non-determinism and the non-deterministic data is also more compressible, e.g., states change less frequently. When going from a wakeup interval of 64 ms to 512 ms the compressed trace size increases along with an increase in both interrupt and timer log sizes as seen in Figure 3.5(b). Although 512 ms wakes up the radio less frequently, sending a message takes a much longer time – 512 versus 64 ms – during which time more interrupts and timer reads are executed. Plain MHO is clearly the least expensive to log, because it does not need to periodically wake the radio to perform clear channel assessments, and messages are sent in the shortest time. For EM, TARDIS reduces the log rate by 83%, and for Collect the reduction is 78%.

The worst case in terms of greatest rate of log data generation with TARDIS is MHO in the network mode with a wakeup interval of 512 ms in which the log grows at 3.3 KB/s. This means that it will fill up 50% of the TelosB flash (i.e., 50% of 1 MB = 500 KB) in 2.5 minutes. Compare this to the baseline case where 50% of the flash will be utilized for logging in just 35 seconds. 50% is significant because that is when a new checkpoint is taken. The entire RAM of the TelosB is 10KB and can be stored to flash in 191ms. MHO when the network is not active fills 50% of the flash in 83 minutes. This re-emphasizes the point that in a lightly loaded network, far less non-deterministic data is generated and consequently, TARDIS is more lightweight in its operation.

Figure 3.5(b) shows the size in flash of the logged interrupts and the timer, data, and state registers. The classification of registers is based on our observations made in



(a) Average power consumption



(b) CPU duty cycle

Fig. 3.6.: Average power consumption and CPU duty cycle of TARDIS instrumented and unmodified applications.

Section 3.3. The largest component in the case of MHO Network is interrupts which is due to interrupts not being as compressible as the registers as shown in Section 3.3. A heavily loaded network exacerbates the issue because it reduces the opportunity for sleep compression explained in Section 3.3.5.

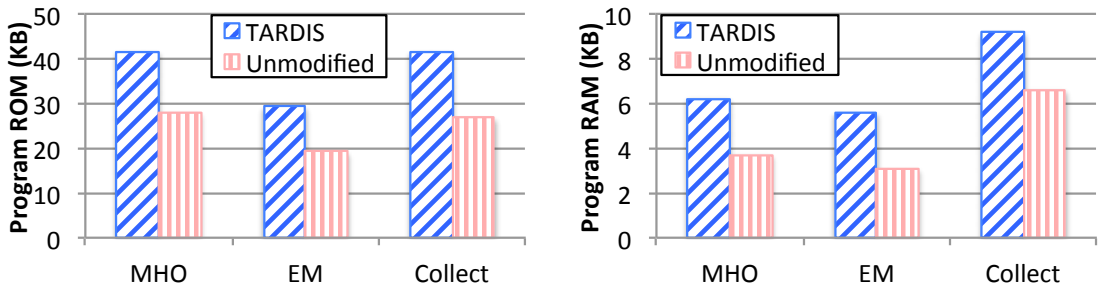
Energy Overhead

The average power consumption of a TARDIS instrumented application and the respective unmodified application are shown in Figure 3.6(a). When the application is not using LPL, there is less than 1% increase in average power consumption between an unmodified application and a TARDIS instrumented application. However, with LPL enabled, the increase in power consumption is between 19% and 146%. Programing a page (256 Bytes) into flash consumes 45mW (a relative power hog) but it only takes 1.5 ms (a relatively short period). The results show that the flash itself is not what is consuming significant power. Instead it is the time taken to record interrupts and reads, along with the time to write to the flash, that keeps the radio active longer, and reduces the energy savings of LPL. This can be seen by the increase in power consumption by TARDIS when going from 512 ms to 64 ms, there are 8 times as many radio wake-ups for channel assessment at 64 ms and TARDIS is keeping the radio awake longer due to logging. Future work could be directed at deferring encoding and flash write operations until the radio returns to sleep. This relies on having large enough buffers that can accommodate all the data until it is time to write the buffer contents to flash.

CPU Overhead

The duty cycle, the fraction of time the CPU is active, is shown in Figure 3.6(b). Unmodified MHO has a higher duty cycle for 512 ms than 64 ms because of the longer time to send messages. As explained with energy overhead, TARDIS keeps the node awake longer when the radio wakes up for channel assessment which explains the higher duty cycle for 64 ms with TARDIS.

CPU time could be reduced by using DMA to transfer log pages to flash. In the current implementation, the CPU is used to perform transfers.



(a) Program size

(b) RAM size

Fig. 3.7.: TARDIS memory overhead in terms of program binary size and statically allocated RAM size.

3.4.3 Static Overhead

The program binary size, shown in Figure 3.7(a), increases due to the TARDIS runtime system code and the instrumentation of the reads and interrupts. In the target WSN OSes TinyOS and Contiki, there is a single image on the node that executes. This single image consists of both the system code and the application code and thus our program binary size refers to the size of this single image. The TARDIS instrumentation of the code consists of inserting calls to the logger for loop counting, interrupts, and reads from peripheral registers. The increase in size for the tested applications range from 23 to 25%, and they fit within the MSP430's 48 KB of program memory.

Figure 3.7(b) shows the statically allocated RAM both with and without TARDIS instrumentation. Only statically allocated RAM is shown because TARDIS does not use dynamically allocated RAM. The increase in statically allocated RAM is due to buffers, and the internal data structures used in compression. TARDIS consumes about 2.6 KB of RAM. The MSP430 has a total RAM size of 10 KB. This RAM consumption can be considered significant for some applications. However, note that this consumption is tunable, one can trade off greater flash usage for lesser RAM

usage. If the RAM allocated to the buffers is smaller, then it will fill up quicker and more frequent logging to flash will occur.

3.4.4 Comparison with gzip, S-LZW, and TinyTracer

An obvious question that arises with respect to the contribution of TARDIS is how well a simple compression of the non-deterministic log would perform. We compare TARDIS to the general purpose compression algorithm `gzip` and to the specialized sensor network compression algorithm S-LZW [39]. To be suitable for sensor networks, the compression algorithm should not require a significant amount of RAM. The TelosB has only 10 KB of RAM, while `gzip` uses a 32 KB sliding window, which makes it unsuitable for our application. S-LZW was designed specifically with sensor networks in mind, and uses a dictionary size of 2KB. In comparison, the complete RAM requirements of TARDIS, which include buffers for writing to flash, is 2.6 KB. Figure 3.8 shows how well `gzip` and S-LZW compress the the trace of non-deterministic data (i.e., reads from peripheral registers and interrupt timings). TARDIS had a reduction of log size of 76% and 96% compared to `gzip` and S-LZW when compressing MHO. In the case of Blink, the log of TARDIS is 2.7 times larger than that of `gzip` and 3% smaller than that of S-LZW. Blink is a very simple application, it repeatedly blinks three LEDS at regular intervals of 1 Hz, 2 Hz, and 4 Hz, which makes the trace easily compressible by the generic `gzip`. Importantly `gzip` has a large window size requirement.

Another approach to trace debugging is to record and replay only control flow. Unlike the complete replay provided by TARDIS, control flow cannot reproduce the state of memory. Many types of bugs are difficult to diagnose with only control flow available, for example, corruption to a message or buffer, an out of bounds index, or an illegal pointer value. The latter two are particularly important to μCs which have no hardware memory protection. The control flow only approach was proposed in TinyTracer [11].

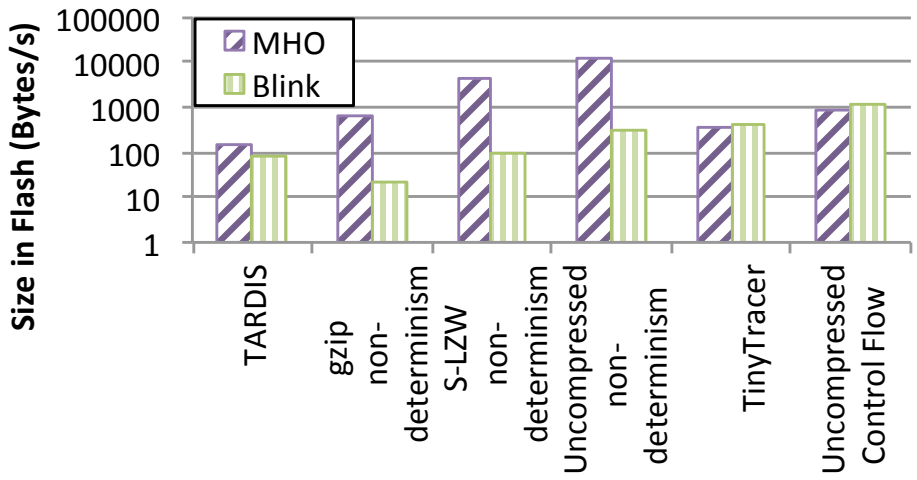


Fig. 3.8.: Size of log in flash for compression methods TARDIS, `gzip`, S-LZW, and TinyTracer. TinyTracer only records control flow.

Figure 3.8 shows the compression results for TinyTracer. "Uncompressed non-determinism" and "uncompressed control flow" are the uncompressed logs for TARDIS and TinyTracer respectively. We observe that the size of trace generated by TARDIS is 79% (for Blink) and 53% (for MHO) smaller than the trace size of TinyTracer. This result was counter-intuitive to us because TARDIS has a more comprehensive set of events that it records. The reason for the log size reduction in TARDIS is that TARDIS records only the non-deterministic inputs whereas TinyTracer records the *effect* of non-deterministic inputs, which is the cascading set of function calls triggered by non-deterministic inputs. TARDIS not only reduces the trace size but also aids in diagnosis of many faults by reproducing the entire execution faithfully including both control and data flow. In contrast, the lack of data flow information in TinyTracer limits the types of faults that can be diagnosed.

3.4.5 CTP Bug Case Study

In this section, we demonstrate how TARDIS can be used to aid in debugging, using a *previously unreported* bug in the Collection Tree Protocol (CTP) as a case study [20]. The bug is triggered when temporary network partition occurs for several seconds due to failure of radio links in the network. The consequence of the bug is that the nodes on the far side of the partition, *i.e.,* on the side away from the base station, are not able to successfully route data messages to a base station for as long as 25 minutes. This is against the principle of CTP which is designed to repair broken routes very quickly — typically within seconds — when data messages need to be delivered.

Description of CTP

CTP is used to collect data in a network by providing anycast communication to a set of root nodes, or base stations. As part of route establishment, all of the nodes broadcast beacons containing a routing metric that represents the expected number of transmissions (ETX) to reach the base station. Each node chooses its best next hop to a base station as the neighbor with the lowest ETX after receiving three consecutive beacons from that neighbor.

CTP differs from previous beacon based approaches in that the rate at which a node sends beacons is dynamic and based on network conditions. Initially, the beacon interval is set to its lowest value of 128 milliseconds. In order to save energy, the beacon interval increases exponentially up to 512 seconds as routes stabilize.

Description of the Bug

We use Figure 3.9 as an example network where the transient failure of the link between nodes 4 and 5 causes the network to become temporarily partitioned. After the network becomes partitioned, the nodes on the far side of the partition (nodes

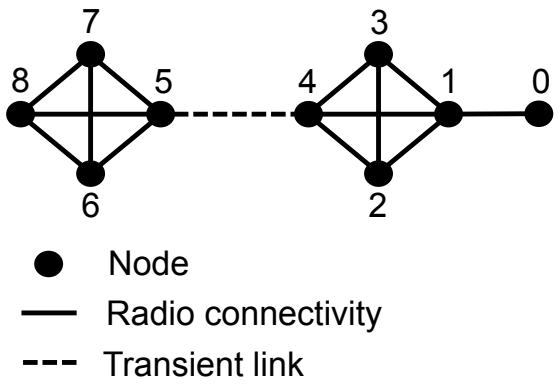


Fig. 3.9.: The radio topology of the network used to study the bug, node 0 is the base station. The bug is triggered when the radio link between nodes 4 and 5 fails for several seconds.

5 through 8) remove their routes to the nodes on the base station side (nodes 0 through 4), and eventually from their routing tables altogether. Nodes 5 through 8 repeatedly choose each other as the next hop neighbor as no route to a base station is present. After the partition is repaired, establishing a route to one of nodes 0 through 4 requires observing at least three beacons from those nodes. The problem is that nodes 0 through 4 are sending beacons at the slowest rate of once every 512 seconds. As a consequence, node 5 will not reestablish node 4 as its next hop neighbor for as long as 1536 seconds ($= 512 \times 3$) or 25 minutes. The reason for slow beacon rate at nodes 0 through 4 is that a good route to the base station has already been established and there is no need to update their routes.

Experimental setup

This bug can be reproduced in lab by creating an artificial network partition by moving the nodes away from the network. In a real network there are many reasons that radio links might fail for several seconds creating network partitions. For example, radio links fail when noise floor is increased by other electronics or when

path loss is created by a temporary high power source obstruction passing between nodes.

To explore this bug, we used a testbed of Telosb nodes in the network layout as shown in Figure 3.9. The nodes are running MultihopOscilloscope and TinyOS 2.1.2. The radio link between nodes 4 and 5 was broken at 30 seconds and re-established at 50 seconds by moving nodes 5 through 8 away from the network. Even though the partition was repaired in 20 seconds, it took over 25 minutes for data messages from nodes 5 through 8 to reach the basestation.

Diagnosing the Bug with Tardis

As described in Section 3.2.5, the typical workflow is for simple invariants to be checked at the basestation. An invariant can require that a message from each node is received within a time period, for example, MultihopOscilloscope expects data every 5 seconds. When the invariant is violated, the basestation disseminates a command to all nodes to not overwrite their traces starting at the current time. It is useful that the dissemination does not depend on the CTP routing protocol. Also dissemination provides eventual delivery, so that when the radio links recover from the failures the command will be delivered.

A key advantage of Tardis is that it can replay the complete state of memory, unlike TinyTracer which only records control flow or Envirolog which must be instructed what values to record [9, 11]. In this case, ETX is a key variable, which can be replayed without any additional instrumentation. ETX is the metric used to decide which node should be the next hop. Figure 3.10 show the ETX value after every call to route update on node 5. At 30 seconds, the increasing value is caused by the partitioned nodes repeatedly choosing each other as the next hop neighbor without any route to a base station actually being present. The continuously rising ETX is a sign that the network has become partitioned, the programmer will want to know duration of the partition and if that is the only cause for the missing data. By

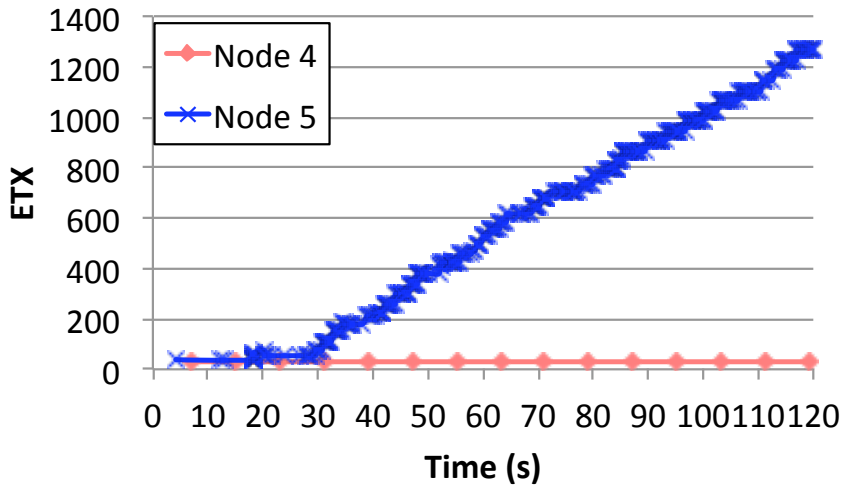


Fig. 3.10.: The ETX values on nodes 4 and 5. Due to the bug, the ETX of node 4 continues to grow even after the link is repaired at 50 seconds.

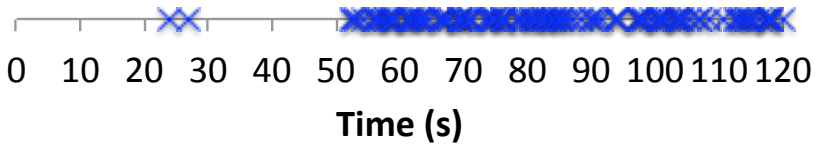


Fig. 3.11.: Beacon messages sent by node 5 and received by node 4.

replaying the nodes and observing the partitioned node's routing tables, it is possible to see that node 5 was connected to node 4 before the partition. This would lead the programmer to node 4. Figure 3.11 shows from the perspective of node 4, all of the beacons received from node 5.

It is clear that the radio link between nodes 4 and 5 was repaired by the 50 second mark, only 20 seconds after that partition began. While the beacon rate is very high at node 5, node 4 is sending beacons at the lowest rate of every 512 seconds. This leads the programmer to conclude that node 4 is missing a condition that will cause

it to reset its beacon interval in this scenario because three beacons from node 4 will help node 5 to find a better path to the base station through node 4.

Bug Fix

The suggested fix is to add an additional condition to reset the beacon interval of CTP. The beacon interval is reset when a beacon is received with an ETX that is significantly larger than the node's own ETX. The intuition is that *nodes within one radio range (neighbors) should not have significantly different ETX values.* The following code is added to the receive beacon function.

```
if (rcvBeacon->etx > routeInfo.etx + 100)
   { call CtpInfo.triggerRouteUpdate(); }
```

We chose ETX difference of 100, representing 10.0 expected retransmissions, because it is much higher than the expected ETX difference between neighbors, which is usually around 20. In this case, resetting the beacon interval (which is done by `triggerRouteUpdate` would cause a node to send beacons at the highest rate of every 128 milliseconds. These beacons would help the distressed neighbors with very high ETX values to pick that node as the next hop almost instantaneously.

Cost of Logging

TARDIS logging rate is low enough that the traces can be collected for a duration much longer than the partition period. The rates of TARDIS log growth for both nodes 4 and 5 are shown in Figure 3.12. During normal operation in the first 30 seconds the logging rate at both nodes is below 1 KB/s. This quickly changes for node 5 when the partition starts at 30 seconds. The increased logging is caused by an increase in radio messages being sent and received. This is because data messages are being fruitlessly forwarded through routing loops, and beacons are being sent at the highest rate by nodes on the far side of the network.

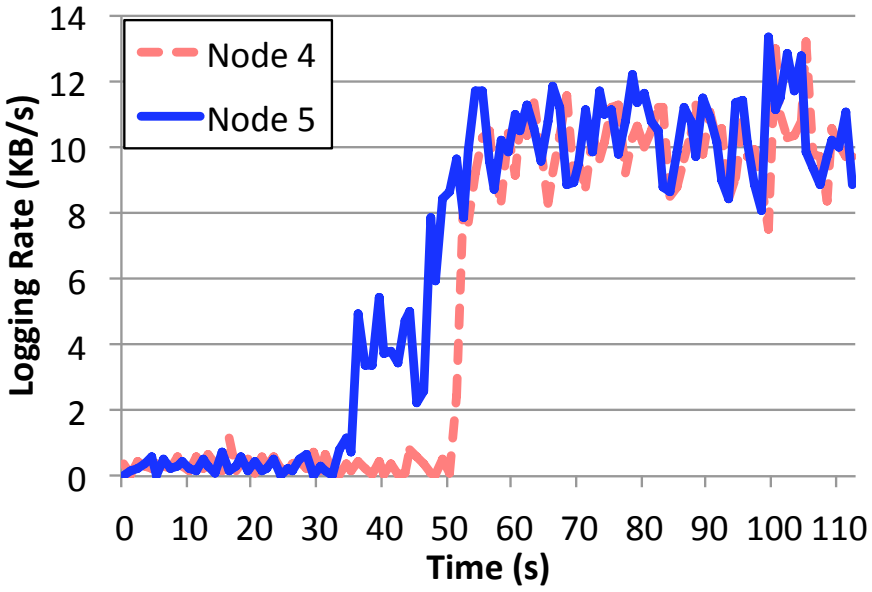


Fig. 3.12.: The rate at which the log grows at nodes 4 and 5. The link between nodes 4 and 5 fails at 30 seconds and returns at 50 seconds.

Node 4 sees an increase in logging only after the partition is repaired at 50 seconds. The high logging rate is caused by the beacon messages received from node 5.

3.5 Discussion

There are some limitations in the current implementation of TARDIS CIL. Expressions containing a read from a peripheral register can only be identified if the register is addressed with a constant. This is the typical method for addressing registers because they are at fixed locations in memory. One exception is the ADC data registers which form a 15 word array of registers. TARDIS CIL could be modified to instrument all instructions with a memory reference where the base is a constant equal to a peripheral register address. Another limitation is TARDIS CIL cannot identify reads from DMA. DMA can be used to transfer peripheral register values to memory, resulting in memory containing non-deterministic values. A solution would be for

TARDIS CIL to instrument the DMA interrupt handler with code that discovers the range of memory written to and transfers the block of memory to the logger.

Our current work focuses on the replay of a single node. Messages received by the node are faithfully reproduced from the captured non-determinism. However, for bugs which are manifested through the interaction of multiple nodes, it is useful to replay nodes in a consistent manner, meaning no message receives are replayed before their corresponding message sends. The standard method is to use Lamport clocks and has been illustrated for distributed record and replay by liblog [29]. A recently proposed lightweight causality tracking technique, CADeT, is specifically designed for resource constrained WSNs [40]. This technique only requires recording a couple of additional counters per message to the log and could easily be applied to TARDIS.

3.6 Related Work

We structure our discussion of related work in three categories — (1) record and replay of single nodes (on desktop class machines), (2) record and replay in distributed applications, and (3) WSN debugging.

3.6.1 Replay of Single Nodes

One class of solutions deals with multiprocessor machines and how to handle the non-determinism introduced by different processes running on the different processors on the same machine. The challenges are to determine what needs to be logged — the complete logging involves assigning a global order to all shared memory accesses and this incurs a 10-100× runtime overhead [41]. Some recent techniques [42] make the observation that the thread access orders of shared memory locations can be recorded cheaply with support from static analysis. R2 [43] allows developers to choose which application functions to record. Our work is simplified by not considering the added burden of non-determinism introduced by multiple core and processor systems. This

is justified by the rarity of multiple core μC used in WSN and energy conscious embedded applications.

3.6.2 Replay of Distributed Applications

The solutions in this category deal with replaying applications that have multiple components that exchange messages among themselves. The primary concern is to faithfully reproduce the global state from the local states and the message exchanges. The primary systems in this category are `liblog` [29], Friday [44], and iTarget [45].

`liblog` is an application level library which intercepts calls to libc and logs their results. Friday builds on top of `liblog` and provides a system that can track arbitrary global invariants at the fine granularity of source symbols. iTarget decides on a replay interface for the application so that its interactions with other software elements can be faithfully recorded.

In contrast to the above line of work, we focus on tracing events of a single node. Our work could be extended to replay of multiple nodes using the techniques described above. Specifically, this implies tracking causality across nodes through message sends and receives.

3.6.3 WSN Debugging

The works in this category can be sub-divided into synchronous and asynchronous debugging. In synchronous debugging, the developer interacts with the application while the application is running and tries to debug any problem as it arises [46, 47]. Minerva [48] connects a debug board to each node in the network. The debug boards use the μCs JTAG port to enable stopping all nodes simultaneously to take snapshots of memory and collecting traces of the node's state while they are running.

In asynchronous debugging, information is collected at runtime and used for offline debugging. TARDIS falls into the class of asynchronous debugging. Within asynchronous debugging, some techniques rely on a model checking approach [4] while

most rely on collecting runtime information and deducing anomalous behavior automatically by mining patterns in the runtime information [7, 10, 11]. The record and replay approaches for WSNs are most closely related to our current work.

Envirolog [9] allows a developer to specify events (e.g., function calls or variable updates) at any layer in code to be captured during a record phase and then reproduced during a replay phase. TARDIS is different from Envirolog in three ways. First, Envirolog cannot reproduce all race conditions. Envriolog uses timestamps to reproduced the timing of events. Given the limitations of the TinyOS clock and timer modules it is only able to deliver events with millisecond precision. This may result in missed race condition bugs, because events are delivered thousands of cycles differently from when they were recorded. TARDIS is able to reproduce all race conditions because it delivers events with the precision of a single instruction by recording the PC value and cycle count. Second, Envirolog does not explore recording a sufficient set of non-deterministic events necessary for complete and consistent replay at the system level. It is not sufficient to record only sensor readings and radio send and receive function calls. For example, if a node receives a command to change its sleep cycle, that command must be reproduced during replay or the recorded log may contain events that occur when the node is asleep during replay. Finally, Envirolog does not explore compression of logged events. If Envirolog were setup to log all non-determinism, then it would be comparable to the baseline case of where no compression is used.

Aveksha [7] uses extra hardware to record traces from the μC JTAG port without interfering with the execution of the node. The events that can be recorded are limited by the bandwidth of the JTAG port, for example, function entry and exit points but not complete control flows. Minerva [48] also uses the JTAG port to collect runtime traces, which enables it to also be used for asynchronous debugging. TinyTracer [11] records the control flow both within functions and across functions. FlashBox [16] is similar in its goals to our work. It adds a compiler pass which instruments code to record non-deterministic information, specifically the execution timeline of interrupts.

The approach requires modified hardware: an additional μC and flash are dedicated to logging. The recorded information only allows a replay of the timeline of interrupts. Prius [49] is a software solution for compressing control flow traces. It relies on offline training to learn what are the common control flow patterns and then compressing runtime trace segments that match against these patterns. We could use it as part of TARDIS, provided we can identify *a priori* common patterns. Also, Prius' reliance on offline training with representative traces raises the bar on its adoption. These existing solutions do not provide comprehensive system-level replay, i.e., replay that is able to reproduce both control flow at an instruction level and the state of memory at any point in time. Using these techniques requires knowing in advance where a bug is likely to manifest itself and be diagnosable.

3.7 Conclusion

TARDIS is the first general-purpose software-only record and replay implementation for WSNs. Our technique supports a *complete* re-execution of the code, thereby enabling the use of many other debugging tools during replay. We have designed and implemented TARDIS, which consists of a compiler plugin, a runtime component, and a replay component. We have made seven key observations common to sensor networks that enable record and replay. The current implementation targets the MSP430 μC, however it is generalizable to other architectures by creating new register definition files.

4. CONCLUSION

This thesis is guided by the principle:

> *Maintain as much detailed information as possible about the execution of WSNs in deployment.*

Two alternative approaches to achieving this goal have been explored, including hardware and software-only. In Aveksha we demonstrated a hardware based approach that uses off-the-self components to non-intrusively record software execution. In TARDIS we demonstrated a software based approach that provided system-level record and replay by recording all sources of non-determinism in an efficient manner.

REFERENCES

REFERENCES

[1] J. Eriksson, F. Osterlind, T. Voigt, N. Finne, S. Raza, N. Tsiftes, and A. Dunkels, "Demo abstract: Accurate power profiling of sensornets with the COOJA/MSPsim simulator," in *Mobile Adhoc and Sensor Systems, 2009. MASS '09. IEEE 6th International Conference on*, Oct 2009, pp. 1060–1061. [Online]. Available: http://dx.doi.org/10.1109/MOBHOC.2009.5337011

[2] J. Eriksson, F. Österlind, N. Finne, N. Tsiftes, A. Dunkels, T. Voigt, R. Sauter, and P. J. Marrón, "Towards interoperability testing for wireless sensor networks with COOJA/MSPSim," in *Proceedings of the 6th European Conference on Wireless Sensor Networks, EWSN'09*, 2009.

[3] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr, "Efficient memory safety for tinyos," in *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, ser. SenSys '07. New York, NY, USA: ACM, 2007, pp. 205–218. [Online]. Available: http://doi.acm.org/10.1145/1322263.1322283

[4] P. Li and J. Regehr, "T-check: Bug finding for sensor networks," in *Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks*, ser. IPSN '10. New York, NY, USA: ACM, 2010, pp. 174–185. [Online]. Available: http://doi.acm.org/10.1145/1791212.1791234

[5] H. Thane and H. Hansson, "Using deterministic replay for debugging of distributed real-time systems," in *Proceedings of the 12th Euromicro Conference on Real-time Systems*, ser. Euromicro-RTS'00. Washington, DC, USA: IEEE Computer Society, 2000, pp. 265–272. [Online]. Available: http://dl.acm.org/citation.cfm?id=1947412.1947455

[6] G. Barrenetxea, F. Ingelrest, G. Schaefer, and M. Vetterli, "The hitchhiker's guide to successful wireless sensor network deployments," in *Proceedings of the 6th ACM conference on Embedded network sensor systems*, ser. SenSys '08. New York, NY, USA: ACM, 2008, pp. 43–56. [Online]. Available: http://doi.acm.org/10.1145/1460412.1460418

[7] M. Tancreti, M. S. Hossain, S. Bagchi, and V. Raghunathan, "Aveksha: A hardware-software approach for non-intrusive tracing and profiling of wireless embedded systems," in *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '11. New York, NY, USA: ACM, 2011, pp. 288–301. [Online]. Available: http://doi.acm.org/10.1145/2070942.2070972

[8] H. Thane, D. Sundmark, J. Huselius, and A. Pettersson, "Replay debugging of real-time systems using time machines," in *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 288.2–. [Online]. Available: http://dl.acm.org/citation.cfm?id=838237.838487

[9] 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 *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, April 2006, pp. 1–14. [Online]. Available: http://dx.doi.org/10.1109/INFOCOM.2006.114

[10] M. Wang, Z. Li, F. Li, X. Feng, S. Bagchi, and Y.-H. Lu, "Dependence-based multi-level tracing and replay for wireless sensor networks debugging," in *Proceedings of the 2011 SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems*, ser. LCTES '11. New York, NY, USA: ACM, 2011, pp. 91–100. [Online]. Available: http://doi.acm.org/10.1145/1967677.1967691

[11] V. Sundaram, P. Eugster, and X. Zhang, "Efficient diagnostic tracing for wireless sensor networks," in *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '10. New York, NY, USA: ACM, 2010, pp. 169–182. [Online]. Available: http://doi.acm.org/10.1145/1869983.1870001

[12] T. Stathopoulos, D. Mclntire, and W. Kaiser, "The energy endoscope: Real-time detailed energy accounting for wireless sensor nodes," in *Information Processing in Sensor Networks, 2008. IPSN '08. International Conference on*, April 2008, pp. 383–394.

[13] R. Fonseca, P. Dutta, P. Levis, and I. Stoica, "Quanto: Tracking energy in networked embedded systems," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 323–338. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855764

[14] "Green hills software inc," http://www.ghs.com/.

[15] X. Jiang, P. Dutta, D. Culler, and I. Stoica, "Micro power meter for energy monitoring of wireless sensor networks at scale," in *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, ser. IPSN '07. New York, NY, USA: ACM, 2007, pp. 186–195. [Online]. Available: http://doi.acm.org/10.1145/1236360.1236386

[16] S. Choudhuri and T. Givargis, "Flashbox: A system for logging non-deterministic events in deployed embedded systems," in *Proceedings of the 2009 ACM Symposium on Applied Computing*, ser. SAC '09. New York, NY, USA: ACM, 2009, pp. 1676–1682. [Online]. Available: http://doi.acm.org/10.1145/1529282.1529657

[17] "IAR Embedded Workbench for TI MSP430," http://www.iar.com.

[18] "Monsoon inc. power monitor," http://www.msoon.com/LabEquipment/PowerMonitor/.

[19] P. Dutta, M. Feldmeier, J. Paradiso, and D. Culler, "Energy metering for free: Augmenting switching regulators for real-time monitoring," in *Proceedings of the 7th International Conference on Information Processing in Sensor Networks*, ser. IPSN '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 283–294. [Online]. Available: http://dx.doi.org/10.1109/IPSN.2008.58

[20] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis, "Collection tree protocol," in *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '09. New York, NY, USA: ACM, 2009, pp. 1–14. [Online]. Available: http://doi.acm.org/10.1145/1644038.1644040

[21] M. Maróti, B. Kusy, G. Simon, and A. Lédeczi, "The flooding time synchronization protocol," in *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*, ser. SenSys '04. New York, NY, USA: ACM, 2004, pp. 39–49. [Online]. Available: http://doi.acm.org/10.1145/1031495.1031501

[22] M. Hossain, A. Alim Al Islam, M. Kulkarni, and V. Raghunathan, "μSETL: A set based programming abstraction for wireless sensor networks," in *Information Processing in Sensor Networks (IPSN), 2011 10th International Conference on*, April 2011, pp. 354–365.

[23] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Local Computer Networks, 2004. 29th Annual IEEE International Conference on*, Nov 2004, pp. 455–462.

[24] R. Huber, P. Sommer, and R. Wattenhofer, "Demo abstract: Debugging wireless sensor network simulations with YETI and COOJA," in *IPSN*, 2011.

[25] T. Goodspeed, "Goodfet," http://goodfet.sourceforge.net, 2010.

[26] K. Shankar and R. Lysecky, "Control Focused Soft Error Detection for Embedded Applications," *Embedded Systems Letters, IEEE*, vol. 2, no. 4, pp. 127–130, 2010.

[27] H. A. Nguyen, A. Forster, D. Puccinelli, and S. Giordano, "Sensor node lifetime: An experimental study," in *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2011 IEEE International Conference on*, March 2011, pp. 202–207.

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

[29] D. Geels, G. Altekar, S. Shenker, and I. Stoica, "Replay debugging for distributed applications," in *Proceedings of the Annual Conference on USENIX '06 Annual Technical Conference*, ser. ATEC '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 27–27. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267359.1267386

[30] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging operating systems with time-traveling virtual machines," in *Proceedings of ATEC '05*. USENIX Association, 2005, pp. 1–1. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247360.1247361

[31] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: Intermediate language and tools for analysis and transformation of c programs," in *Proceedings of the 11th International Conference on Compiler Construction*, ser. CC'02. Springer-Verlag, 2002, pp. 213–228.

[32] J. L. Hill and D. E. Culler, "Mica: A wireless platform for deeply embedded networks," *IEEE Micro*, vol. 22, no. 6, pp. 12–24, Nov. 2002.

[33] J. Polastre, J. Hill, and D. Culler, "Versatile low power media access for wireless sensor networks," in *Proceedings of the 2Nd International Conference on Embedded Networked Sensor Systems*, ser. SenSys '04. New York, NY, USA: ACM, 2004, pp. 95–107. [Online]. Available: http://doi.acm.org/10.1145/1031495.1031508

[34] J. Polastre, R. Szewczyk, and D. Culler, "Telos: enabling ultra-low power wireless research," in *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, April 2005, pp. 364–369.

[35] "LZRW1," http://www.ross.net/compression/lzrw1.html.

[36] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *Information Theory, IEEE Transactions on*, vol. 23, no. 3, pp. 337–343, May 1977.

[37] R. Tan, G. Xing, J. Chen, W.-Z. Song, and R. Huang, "Quality-driven volcanic earthquake detection using wireless sensor networks," in *Real-Time Systems Symposium (RTSS), 2010 IEEE 31st*, Nov 2010, pp. 271–280.

[38] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh, "Fidelity and yield in a volcano monitoring sensor network," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 381–396. [Online]. Available: http://dl.acm.org/citation.cfm?id=1298455.1298491

[39] C. M. Sadler and M. Martonosi, "Data compression algorithms for energy-constrained devices in delay tolerant networks," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, ser. SenSys '06. New York, NY, USA: ACM, 2006, pp. 265–278. [Online]. Available: http://doi.acm.org/10.1145/1182807.1182834

[40] V. Sundaram and P. Eugster, "Lightweight message tracing for debugging wireless sensor networks," in *Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on*, June 2013, pp. 1–12.

[41] T. LeBlanc and J. Mellor-Crummey, "Debugging parallel programs with instant replay," *Computers, IEEE Transactions on*, vol. C-36, no. 4, pp. 471–482, April 1987.

[42] J. Huang, P. Liu, and C. Zhang, "Leap: Lightweight deterministic multi-processor replay of concurrent java programs," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10. New York, NY, USA: ACM, 2010, pp. 207–216. [Online]. Available: http://doi.acm.org/10.1145/1882291.1882323

[43] Z. Guo, X. Wang, J. Tang, X. Liu, Z. Xu, M. Wu, M. F. Kaashoek, and Z. Zhang, "R2: An application-level kernel for record and replay," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 193–208. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855741.1855755

[44] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica, "Friday: Global comprehension for distributed replay," in *Proceedings of the 4th USENIX*

Conference on Networked Systems Design & Implementation, ser. NSDI'07. Berkeley, CA, USA: USENIX Association, 2007, pp. 21–21. [Online]. Available: http://dl.acm.org/citation.cfm?id=1973430.1973451

[45] M. Wu, F. Long, X. Wang, Z. Xu, H. Lin, X. Liu, Z. Guo, H. Guo, L. Zhou, and Z. Zhang, "Language-based replay via data flow cut," in *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE '10. New York, NY, USA: ACM, 2010, pp. 197–206. [Online]. Available: http://doi.acm.org/10.1145/1882291.1882322

[46] 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 *Information Processing in Sensor Networks, 2006. IPSN 2006. The Fifth International Conference on*, 2006, pp. 416–423.

[47] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse, "Clairvoyant: A comprehensive source-level debugger for wireless sensor networks," in *Proceedings of the 5th International Conference on Embedded Networked Sensor Systems*, ser. SenSys '07. New York, NY, USA: ACM, 2007, pp. 189–203. [Online]. Available: http://doi.acm.org/10.1145/1322263.1322282

[48] P. Sommer and B. Kusy, "Minerva: Distributed tracing and debugging in wireless sensor networks," in *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '13. New York, NY, USA: ACM, 2013, pp. 12:1–12:14. [Online]. Available: http://doi.acm.org/10.1145/2517351.2517355

[49] V. Sundaram, P. Eugster, and X. Zhang, "Prius: Generic hybrid trace compression for wireless sensor networks," in *Proceedings of the 10th ACM Conference on Embedded Network Sensor Systems*, ser. SenSys '12. New York, NY, USA: ACM, 2012, pp. 183–196. [Online]. Available: http://doi.acm.org/10.1145/2426656.2426675

VITA

VITA

Matthew Tan Creti received both the BSE in Electrical Engineering and the BS in Computer Science from the University of Iowa in 2005. Matthew enrolled as a MS student in the School of Electrical and Computer Engineering at Purdue University in Fall 2006. Under the supervision of Prof. Saurabh Bagchi, he received the MS degree in Fall 2008. In Spring 2009, he enrolled in the PhD program in the School of Electrical and Computer Engineering at Purdue University, with Dr. Bagchi as his advisor. During his PhD studies he worked as a research assistant in Dr. Bagchi's lab and as an intern for Raytheon BBN Technologies in Cambridge, Massachusetts. His research interests include wireless embedded systems and dependable distributed systems. He is currently a co-founder and the Chief Technology Officer of SensorHound, Inc.