TARDIS: Software-Only System-Level Record and Replay in Wireless Sensor Networks

Matthew Tancreti[1], Vinaitheerthan Sundaram[2], Saurabh Bagchi[1,2], and Patrick Eugster[2,3,*]

[1]School of Electrical and Computer Engineering, Purdue University

[2]Department of Computer Science, Purdue University

[3]Department of Computer Science, TU Darmstadt

ABSTRACT

Wireless sensor networks (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 sensor nodes, the non-determinism of execution that can make it difficult to replicate a buggy run, and the limited hardware resources available on a node. In particular, existing solutions to *record and replay* debugging in WSNs fail to capture the complete code execution, thus negating the possibility of a faithful replay and causing a large class of bugs to go unnoticed. In short, record and replay logs a trace of predefined events while a deployed application is executing, enabling replaying of events later using debugging tools. Existing recording methods fail due to the many sources of non-determinism and the scarcity of resources on nodes.

In this paper we introduce Trace And Replay Debugging In Sensornets (TARDIS), a software-only approach for deterministic record and replay of 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. Despite their domain-specific nature, the techniques presented are applicable to the broader class of resource-constrained embedded systems. We empirically demonstrate the viability of our approach and its effectiveness in diagnosing a newly discovered bug in a widely used routing protocol.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging

Keywords

Wireless Sensor Networks, Tracing, Debugging, Replay

* P. Eugster is partially supported by the German Research Foundation (DFG) under project MAKI ("Multi-mechanism Adaptation for the Future Internet").

IPSN '15, April 14 - 16, 2015, Seattle, WA, USA

http://dx.doi.org/10.1145/2737095.2737096

1 Introduction

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 [20, 6], safe code enforcement [4], and formal testing [16] 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.

1.1 Record and Replay

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 [30]. 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.

1.2 Challenges of Record and Replay in WSNs

However, 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. Additionally, storing data to flash is energy expensive, with frequent flash usage reducing a node's lifetime by a factor of 3 [19].

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 [2] and Contiki [5]. 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 [10]. Manual modification of OS drivers or system libraries, as done in previous record and replay systems [8, 14], 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 [2] and Contiki [5] 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` [8]. We call this *system-level* record and replay, which is more expansive in scope than application-level record and replay.

1.3 TARDIS Approach

In this paper, 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-determinism 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 [26]) or only a predetermined subset of variables and events (e.g., EnviroLog [17]).

1.4 Contributions

This paper 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 record 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 [26], 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

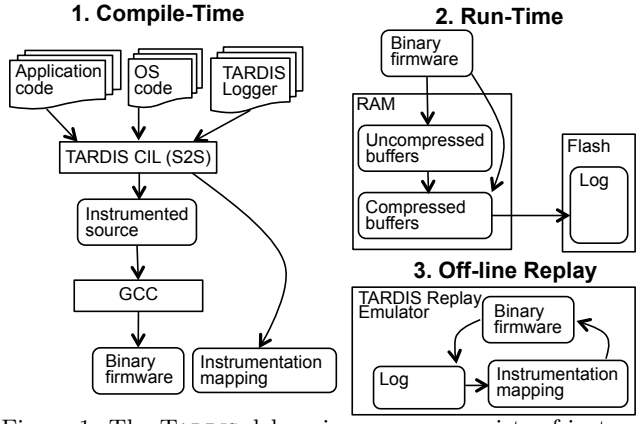


Figure 1: The TARDIS debugging process consists of instrumentation at compile-time, *in situ* logging of trace data at run-time, and off-line replay during debugging.

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.

2 High-Level Design and Implementation

This section introduces the design and implementation of TARDIS.

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

2.2 Compile Time

2.2.1 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.

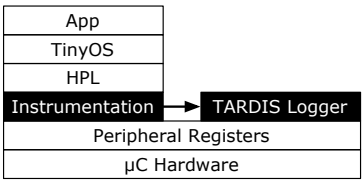


Figure 2: TARDIS instrumentation and logger with respect to the TinyOS stack.

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 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) [18]. 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 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` [8]. 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 3.4 and 3.3 how the identification of masking of peripheral register values and polling loops can help to reduce or completely eliminate logging requirements.

2.2.2 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 [14].

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 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.

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 [6]) 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.

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.

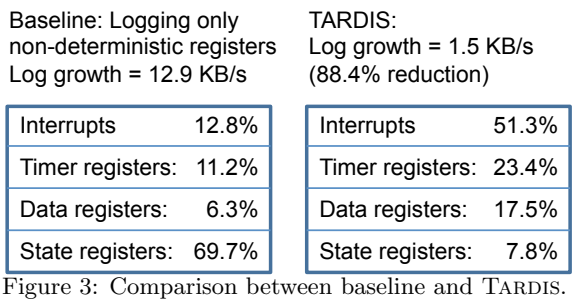


Figure 3: Comparison between baseline and TARDIS.

3 Encoding and Compression of Non-Deterministic Data

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

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 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 3.2. 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 3.2-3.5) and encoding and compression (Sections 3.6-3.8). 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 1 presents an overview of

Table 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

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. The evaluation section shows the overall benefit of TARDIS, with all these techniques operating together, for a wider variety of applications.

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 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.

3.4 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.

3.5 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 [12, 21, 22]. 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 2.2.2 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.

3.6 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 3.5.

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%.

3.7 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.

3.8 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 [1], 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>
```

Figure 4: Logging format.

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 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 3.4. 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 3.6. 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.

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 [26] whilst enabling diagnosis of a much wider class of bugs than TinyTracer. Finally, we give the case study of diagnosing a previously unreported bug.

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 [28], 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 [32]. We run EM in a single hop mode with each node sending directly to a base node.

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.

4.2.1 Flash Size

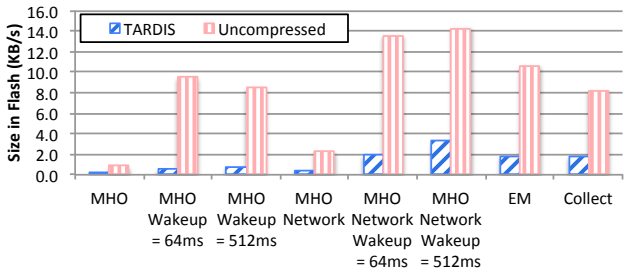
Figure 5(a) shows the rate of log growth for Tardis and for an uncompressed trace. We see that for MHO single node, the log size is reduced by 92%, 94%, and 90% compared to the uncompressed traces for the various wakeup intervals—no LPL, 64 ms, and 512 ms respectively. For MHO Network mode, the reduction 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 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.

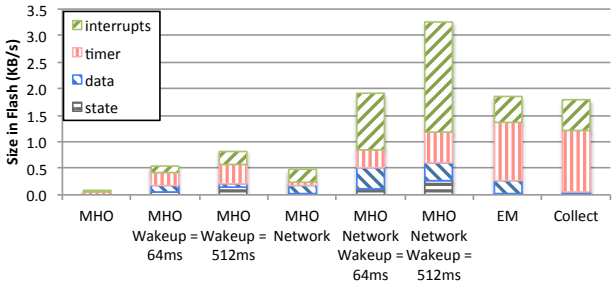
Figure 5(b) shows the size in flash of the logged interrupts and the timer, data, and state registers. The classification of registers is based on our observations made in Section 3. The largest component in the case of MHO Network is interrupts which is due to interrupts not being as compressible as the registers as shown in Section 3. A heavily loaded network exacerbates the issue because it reduces the opportunity for sleep compression explained in Section 3.5.

4.2.2 Energy Overhead

The average power consumption of a Tardis instrumented application and the respective unmodified application are shown in Figure 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 wakeups 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.

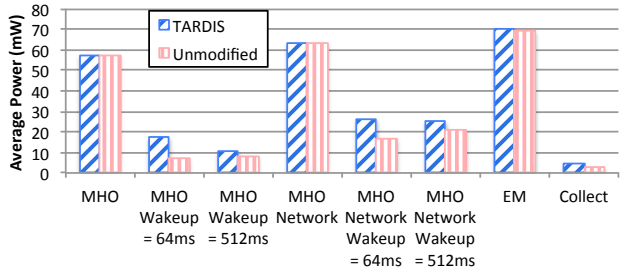


(a) Rate of log growth for TARDIS and uncompressed.

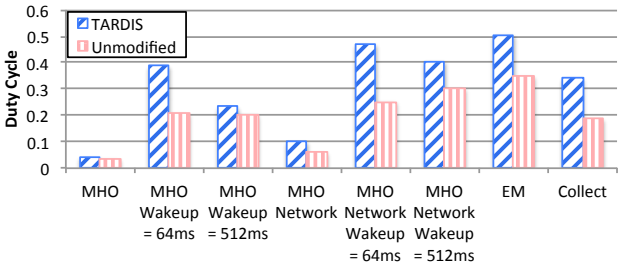


(b) Size of different log components for TARDIS

Figure 5: Rate of log growth and the size of different log components for TARDIS. Uncompressed log rate shown for comparison.



(a) Average power consumption



(b) CPU duty cycle

Figure 6: Average power consumption and CPU duty cycle of TARDIS instrumented and unmodified applications.

4.2.3 CPU Overhead

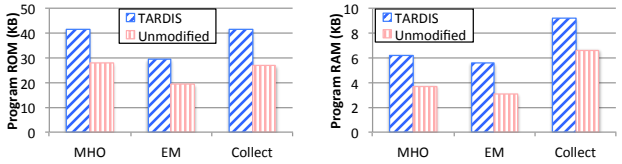
The duty cycle, the fraction of time the CPU is active, is shown in Figure 6(b). Unmodified MHO has a higher duty cycle for 512 ms than 64 ms because of the longer time to send messages. As explained with energy overhead, TARDIS keeps the node awake longer when the radio wakes up for channel assessment 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.

4.3 Static Overhead

The program binary size, shown in Figure 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 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.



(a) Program size (b) RAM size

Figure 7: TARDIS memory overhead in terms of program binary size and statically allocated RAM size.

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 [23]. 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 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 repeat-

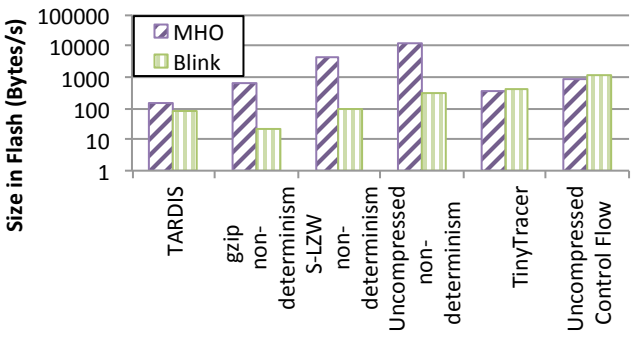


Figure 8: Size of log in flash for compression methods TARDIS, `gzip`, S-LZW, and TinyTracer. TinyTracer only records control flow.

edly 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 [26].

Figure 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.

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 [9]. 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.

4.5.1 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.

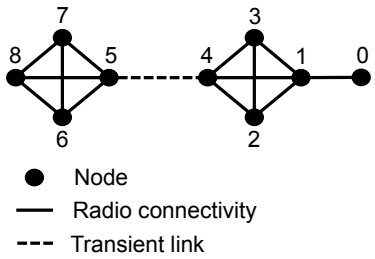


Figure 9: The radio topology of the network used to study the bug, node 0 is the base station. The bug is triggered when the radio link between nodes 4 and 5 fails for several seconds.

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.

4.5.2 Description of the Bug

We use Figure 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 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.

4.5.3 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 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 reestablished 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.

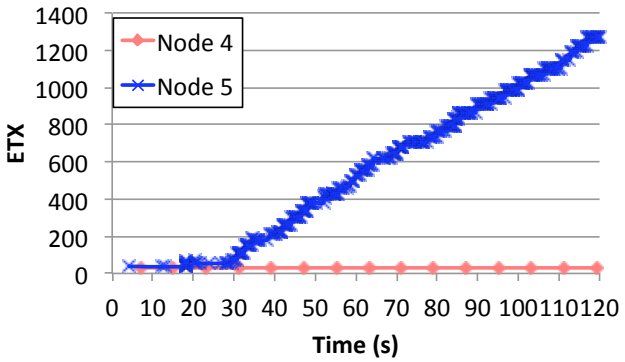


Figure 10: The ETX values on nodes 4 and 5. Due to the bug, the ETX of node 4 continues to grow even after the link is repaired at 50 seconds.

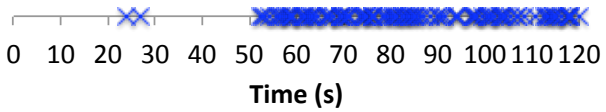


Figure 11: Beacon messages sent by node 5 and received by node 4.

4.5.4 *Diagnosing the Bug with* TARDIS

As described in Section 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 [26, 17]. 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 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 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 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.

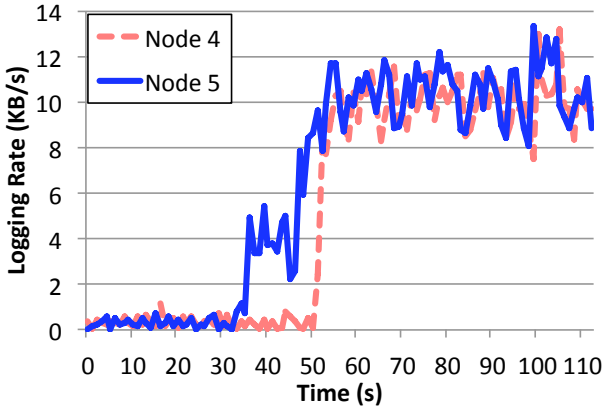


Figure 12: The rate at which the log grows at nodes 4 and 5. The link between nodes 4 and 5 fails at 30 seconds and returns at 50 seconds.

4.5.5 *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.

4.5.6 *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 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.

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.

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 [8]. A recently proposed lightweight causality tracking technique, CADeT, is specifically designed for resource constrained WSNs [25]. This technique only requires recording a couple of additional counters per message to the log and could easily be applied to TARDIS.

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.

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 [15]. Some recent techniques [13] make the observation that the thread access orders of shared memory locations can be recorded cheaply with support from static analysis. R2 [11] 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.

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` [8], Friday [7], and iTarget [34].

`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.

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 [33, 35]. Minerva [24] 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 [16] while most rely on collecting runtime information and deducing anomalous behavior automatically by mining patterns in the runtime information [26, 29, 31]. The record and replay approaches for WSNs are most closely related to our current work.

Envirolog [17] 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 [29] 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 [24] also uses the JTAG port to collect runtime traces, which enables it to also be used for asynchronous debugging. TinyTracer [26] records the control flow both within functions and across functions. FlashBox [3] 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 [27] 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.

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.

8 Acknowledgments

We thank the anonymous referees and our shepherd, Kay Römer. We thank 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. 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.

9 References

[1] LZRW1. http://www.ross.net/compression/lzrw1.html.

[2] TinyOS. http://www.tinyos.net/.

[3] S. Choudhuri and T. Givargis. FlashBox: A system for logging non-deterministic events in deployed embedded systems. In *Proc. of SAC*. ACM, 2009.

[4] N. Cooprider, W. Archer, E. Eide, D. Gay, and J. Regehr. Efficient memory safety for TinyOS. In *Proc. of SenSys*. ACM, 2007.

[5] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - A lightweight and flexible operating system for tiny networked sensors. In *Proc. of LCN*. IEEE, 2004.

[6] 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 *Proc. of EWSN*. Springer, 2009.

[7] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. Friday: Global comprehension for distributed replay. In *Proc. of NSDI*. USENIX, 2007.

[8] D. Geels, G. Altekar, S. Shenker, and I. Stoica. Replay debugging for distributed applications. In *Proc. of ATEC*. USENIX, 2006.

[9] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection tree protocol. In *Proc. of SenSys*. ACM, 2009.

[10] T. Goodspeed. Goodfet. http://goodfet.sourceforge.net.

[11] 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 *Proc. of OSDI*. USENIX, 2008.

[12] J. L. Hill and D. E. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, Nov. 2002.

[13] J. Huang, P. Liu, and C. Zhang. LEAP: Lightweight deterministic multi-processor replay of concurrent Java programs. In *Proc. of FSE*. ACM, 2010.

[14] S. T. King, G. W. Dunlap, and P. M. Chen. Debugging operating systems with time-traveling virtual machines. In *Proc. of ATEC*. USENIX, 2005.

[15] T. LeBlanc and J. Mellor-Crummey. Debugging parallel programs with instant replay. *Computers, IEEE Trans. on*, 100(4):471–482, 1987.

[16] P. Li and J. Regehr. T-check: Bug finding for sensor networks. In *Proc. of IPSN*. ACM, 2010.

[17] L. Luo, T. He, G. Zhou, L. Gu, T. F. Abdelzaher, and J. A. Stankovic. Achieving repeatability of asynchronous events in wireless sensor networks with envirolog. In *Proc. of INFOCOM*. IEEE, 2006.

[18] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. CIL: Intermediate language and tools for analysis and transformation of c programs. In *Proc. of CC*. Springer-Verlag, 2002.

[19] H. Nguyen, A. Forster, D. Puccinelli, and S. Giordano. Sensor node lifetime: An experimental study. In *PERCOM Workshops*. IEEE, 2011.

[20] F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level sensor network simulation with COOJA. In *Proc. of LCN*. IEEE, 2006.

[21] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *Proc. of SenSys*. ACM, 2004.

[22] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proc. of IPSN*. IEEE, 2005.

[23] C. M. Sadler and M. Martonosi. Data compression algorithms for energy-constrained devices in delay tolerant networks. In *Proc. of SenSys*. ACM, 2006.

[24] P. Sommer and B. Kusy. Minerva: Distributed tracing and debugging in wireless sensor networks. In *Proc. of SenSys*. ACM, 2013.

[25] V. Sundaram and P. Eugster. Lightweight message tracing for debugging wireless sensor networks. In *Proc. of DSN*. IEEE, 2013.

[26] V. Sundaram, P. Eugster, and X. Zhang. Efficient diagnostic tracing for wireless sensor networks. In *Proc. of SenSys*. ACM, 2010.

[27] V. Sundaram, P. Eugster, and X. Zhang. Prius: Generic hybrid trace compression for wireless sensor networks. In *Proc. of SenSys*. ACM, 2012.

[28] R. Tan, G. Xing, J. Chen, W.-Z. Song, and R. Huang. Quality-driven volcanic earthquake detection using wireless sensor networks. In *Proc. of RTSS*. IEEE, 2010.

[29] M. Tancreti, M. Hossain, S. Bagchi, and V. Raghunathan. Aveksha: A hardware-software approach for non-intrusive tracing and profiling of wireless embedded systems. In *Proc. of SenSys*. ACM, 2011.

[30] H. Thane and H. Hansson. Using deterministic replay for debugging of distributed real-time systems. In *Proc. of Euromicro-RTS*. IEEE, 2000.

[31] 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 *Proc. of LCTES*. ACM, 2011.

[32] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and yield in a volcano monitoring sensor network. In *Proc. of OSDI*. USENIX, 2006.

[33] 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 *Proc. of IPSN*. ACM, 2006.

[34] 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 *Proc. of FSE*. ACM, 2010.

[35] J. Yang, M. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: A comprehensive source-level debugger for wireless sensor networks. In *Proc. of SenSys*. ACM, 2007.

[36] J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *Information Theory, IEEE Trans. on*, 23(3):337–343, 1977.