Declarative Tracepoints: A Programmable and Application Independent Debugging System for Wireless Sensor Networks

Qing Cao, Tarek Abdelzaher
Department of Computer Science
University of Illinois
{qcao2,zaher}@cs.uiuc.edu

John Stankovic,
Kamin Whitehouse
Department of Computer Science
University of Virginia
{stankovic,whitehouse}@cs.virginia.edu

Liqian Luo
Microsoft Research
liqian@microsoft.com

ABSTRACT

Effective debugging usually involves watching program state to diagnose bugs. When debugging sensor network applications, this approach is often time-consuming and error-prone, not only because of the lack of visibility into system state, but also because of the difficulty to watch the *right* variables at the *right* time. In this paper, we present declarative tracepoints, a debugging system that allows the user to insert a group of action-associated checkpoints, or tracepoints, to applications being debugged at runtime. Tracepoints do not require modifying application source code. Instead, they are written in a declarative, SQL-like language called TraceSQL independently. By triggering the associated actions when these checkpoints are reached, this system automates the debugging process by removing the human from the loop. We show that declarative tracepoints are able to express the core functionality of a range of previously isolated debugging techniques, such as EnviroLog, NodeMD, Sympathy, and StackGuard. We describe the design and implementation of the declarative tracepoints system, evaluate its overhead in terms of CPU slowdown, illustrate its expressiveness through the aforementioned debugging techniques, and finally demonstrate that it can be used to detect real bugs using case studies of three bugs based on the development of the LiteOS operating system.

Categories and Subject Descriptors

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

General Terms

Design, Experimentation, Performance

Keywords

Declarative Tracepoints, Embedded Debugging, Wireless Sensor Networks

1. INTRODUCTION

Effective debugging usually involves watching program state to diagnose abnormal behavior. When debugging sensor networks, observing state is challenging in that it requires watching the *right* set of variables at the *right* time. That set is hard to know in advance. Meanwhile, watching everything is not feasible due to severe resource limitations at individual sensor nodes. Often, a node could crash and restart before useful information is collected, in which case all state is lost. Therefore, the debugging process for sensor network applications remains time-consuming, error-prone, and difficult.

To simplify the debugging process, we present the *declarative tracepoints* (DT) system, which allows the developer to insert a group of action-associated checkpoints at runtime. We refer to these action-associated checkpoints as *tracepoints*, or *probes*, analogous to the way test probes are used in electronic hardware to debug circuits. These tracepoints are programmed in an SQL-like declarative language, called *TraceSQL*. By triggering the associated actions when tracepoints are reached, DT removes the human from the loop, and makes the debugging process programmable.

DT has two key advantages. First, DT does not require modifying application source code. Based on dynamic instrumentation, it enables programmers to add and remove tracepoints at runtime, without requiring application reboots. Such flexibility allows programmers to try out multiple rounds of modifications without the need to re-deploy applications. This is particularly attractive in sensor networks, where recompilation and re-deployment of applications is usually a lengthy, error-prone process. Second, to implement associated actions, DT introduces the TraceSQL language to program the debugging actions. Being programmable, DT can express a wide range of debugging techniques that were previously hardwired for unique application needs. In this sense, DT acts like the *thin waist* of a systematic framework where multiple debugging techniques co-exist. To the best of our knowledge, DT is the first debugging system to simultaneously provide both a declarative programming language and independence from application source code for wireless sensor networks.

The usage model of DT is as follows. First, a user application is deployed into multiple nodes. After observing abnormal behavior, the developer writes DT scripts in TraceSQL, compiles the scripts into Tracepoint Engines (TEs), and installs the TEs to debug the application. A TE inserts tracepoints into application binaries, triggers associated actions when such tracepoints are reached, and exposes information in the form of messages or traces for online or off-line diagnosis. In this sense, the TE is an agent in lieu of the developer, streamlining the debugging process because it no longer involves user interaction. Finally, the developer can uninstall the current TEs and install new ones if the bug has not been resolved. The on-demand deployment of TEs keeps the debugging code base small, thus minimizing the resource consumption of the debugging process.

The design of DT is partially inspired by features offered by Aspect Oriented Programming (AOP) [13]. The design motivation of AOP is to achieve separation of concerns and to avoid tangling code by identifying *cross-cutting aspects* that cut across the system's basic components at *join points*, where additional *advice* is applied. The aspects and components in AOP are usually developed using different languages, and are later weaved together either at compile time or at runtime. For complicated systems, such an approach is promising to improve the isolation, composition, and reuse of software modules.

DT and the TraceSQL language can be viewed as one aspect of sensor network development, *the debugging aspect*. Just like AOP, TraceSQL programs are developed independently of the debugged applications. TraceSQL programs cut across functional components, in the form of Tracepoint Engines (TEs). The associated actions are essentially a form of advice for debugging purposes.

We have implemented a prototype of DT on top of the LiteOS operating system, a recent thread-based operating system developed at the University of Illinois [6]. We chose LiteOS mostly because of its support for interactive control of network behavior through its Unix-like shell, such as the file copy command for easy retrieval of data. Therefore, we do not address trace retrieval separately in the DT system. The prototype we implemented supports instrumentation of both user applications and the LiteOS kernel. We also selected four representative debugging techniques, namely EnviroLog [17], NodeMD [14], Sympathy [20], and StackGuard [8], to demonstrate that they can be expressed with TraceSQL. Finally, we also used DT to retroactively debug the documented bugs in the development version of LiteOS and its applications from October 2007 to March 2008, as well as to solve a problem in the communication stack development based on LiteOS. We present these bugs as representative case studies.

Note that, DT can also be implemented on other sensor network operating systems, as long as the following two features are supported. First, the operating system should support dynamic loading of new modules, so that the compiled TE can be installed incrementally in addition to the deployed user applications. Second, the operating system should support access to non-volatile storage, such as flash. Several existing operating systems meet such requirements, including TinyOS [11] (with TinyThreads [19], Deluge [12], and the Matchbox file system [2] installed), Contiki [9], and Mantis [4]. Porting declarative tracepoints to these operating systems is therefore possible. Also note that while DT is designed for wireless sensor networks, it applies to more generic embedded system debugging as well. Porting DT for these needs is outside the scope of this paper.

The remainder of this paper is organized as follows. Section 2 summarizes the related work. In Section 3, we give an overview of DT. Section 4 presents the details of the TraceSQL language, illustrating the core features using examples. In Section 5, we discuss the implementation of declarative tracepoints. Section 6 presents performance evaluation focusing on overhead. In Section 7, we explore and evaluate the expressiveness of DT using a series of debugging techniques from the literature. Section 8 presents three real bugs based on the LiteOS operating system as case studies to demonstrate the effectiveness of DT. Section 9 concludes the paper and presents directions for future work.

2. RELATED WORK

Debugging has been an active topic probably ever since software has been written. In the area of sensor networks, many debugging techniques have been proposed. In this paper, we use TraceSQL to express three such techniques; namely, EnviroLog [17], NodeMD [14], and Sympathy [20]. We also use TraceSQL to express a more general technique called StackGuard [8]. Below, we describe them in more detail.

The first tool that we use TraceSQL to express is EnviroLog. It aims to improve repeatability of experimental testing of distributed event-driven applications, based on the observation that the system state can change depending on the event sequence and timing. Hence, debugging such applications is complicated by non-repeatable event sequences caused by the dynamic environmental inputs. To address this challenge, EnviroLog provides an event recording and replay service that captures and replays events with the help of the non-volatile flash. Its compiler modifies the application source code such that every time an instrumented function is called, its invoked time and parameters are stored into flash. Later, EnviroLog replays this sequence of events by executing the same sequence of functions with the same parameters. After the code is installed, EnviroLog allows the user to issue `START_RECORD` and `START_REPLAY` commands to record and replay events on demand.

The second tool expressed, NodeMD, is designed to diagnose node-level faults in sensor network applications. It focuses on catching software faults before they completely disable the remote sensor node, so that the user can be provided with diagnostic information to troubleshoot the root cause. It tries to catch three types of bugs: stack overflows, livelocks, and deadlocks, in addition to application-specific faults. It also provides remote retrieval of the logged information stored in a circular buffer, so that the probable root of the bug can be traced.

The third tool, Sympathy, detects and debugs failures by collecting metrics and performing an analysis procedure at the sink to localize the most likely failure source. The key assumption made by Sympathy is that for a broad class of data gathering applications, it is possible to diagnose failures by analyzing a minimal set of metrics at a centralized sink. It traces the failure source to one of the three possibilities: the node itself, the communication path, or the sink.

The last tool we express is StackGuard, a more generic tool for detecting stack corruption caused by buffer overruns (e.g., when the return address of a function is over-written). This problem is also known as the buffer overrun security exploit. Having received intensive attention, this problem is addressed in multiple ways, and StackGuard is one of the better-known techniques in that it virtually eliminates all buffer overflows with the help of the *canary word*. More specifically, StackGuard modifies the generated prologue and epilogue code for functions to insert canary words. The assumption held by StackGuard is that if some code in a function modifies the return address, it must have modified the canary word as well, assuming that the application does not know the value and size of the canary word. By checking the integrity of the canary word, StackGuard can detect malfunctioning code.

Besides the four representative debugging techniques, many other tools have been proposed for different debugging purposes in the sensor network community. Clairvoyant [24], JTAG [1], and the LiteOS shell [6] provide interactive source-level debugging commands such as *break*, *step*, and *watch* to access program state. TOSSIM [16], EmStar [10], and Avrora [21] provide simulation environments for sensor network applications. SNMS [22] provides logging and retrieval of runtime state for fault diagnosis. Some tools recognize visibility as the one of main obstacles for debugging, hence propose to improve visibility in various ways [23]. Tools for improving memory safety also exist [7]. Various virtual machines [15] developed for sensor networks can also be modified to provide robustness and error checking on the nodes. However, none of these tools implement languages specialized for debugging that can express existing techniques. Hence, we believe that DT is complementary to these previous techniques.

Outside the area of sensor networks, one tool that has used dynamic instrumentation for debugging is DTrace [3], a comprehensive dynamic tracing framework developed by Sun Microsystems on Solaris 10. DTrace allows the user to write scripts using the D programming language to perform runtime instrumentation and trace collection. Compared to DTrace, our work is novel in the following two respects. First, DTrace is tightly integrated with the underlying operating system and only supports PCs with sufficient system resources. The design and implementation of our DT system, on the other hand, consists of design choices specific to sensor nodes to fit into their stringent resource limitations. Second, the debugging needs addressed in this paper are very different from those addressed by DTrace scripts. While DTrace primarily addresses problems more specific to its own environment, such as CPU scheduling tracing and I/O activities, our DT system targets debugging tasks of more interest to sensor networks, such as stack overflows and the special need to replay sensor readings. These significant differences in goals and approaches distinguish DT from DTrace.

3. DESIGN

The DT system aims to provide a programmable and application-independent debugging architecture. Figure 1 shows its overall architecture. Below, we identify the major design goals.

3.1 Application Independence

Our first goal is to achieve application independence. More specifically, we hope that DT programs should be developed independently of applications, and should not require

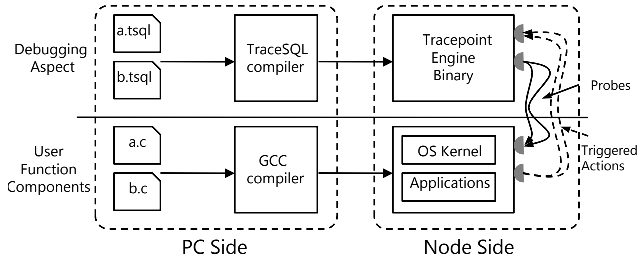


Figure 1: Overall System Architecture of Declarative Tracepoints

source level modifications to the applications under debugging. While this approach may sometimes prevent the compiler from performing better global optimizations, it achieves separation of concerns by isolating debugging code from application code. Hence, debugging can be performed without requiring re-compilations and re-deployments of applications.

DT works as follows. As shown in Figure 1,TraceSQL programs are compiled into TE executables. Once installed, TE inserts tracepoints into applications by instrumenting their binaries. The instruction flow jumps to an action handler whenever a tracepoint is reached. The action handler is carefully designed such that its existence is transparent to user applications except in timing [1]. Indeed, the number of CPU cycles for running a user application is changed inevitably if new functionality is added. However, this is usually not a big concern in that it can be viewed that the application reached a mini context switch at each tracepoint, although the kernel is oblivious. Given that DT is designed for multi-threaded operating systems (each TE is executed as a separate thread), it is reasonable to assume that additional context switches should not lead to extra application bugs in most cases [2].

3.2 Ease of Programming

Our second goal is to allow easy DT programming. To this end, we adopt a declarative language syntax similar to SQL. Declarative languages are well known for their ability to express complicated operations with short programs. The customized language we developed, TraceSQL, focuses on expressing the locations of tracepoints and the associated actions. Below, we show an example that records every context switch caused by applications. In LiteOS, each context switch invokes the `yield()` function in the `syscall.c` file. Therefore, we simply add a tracepoint at the beginning of this function to track context switches, shown as follows. The syntax of TraceSQL is described in Section 4.

```
TRACE yield() FROM syscall.c EXECUTE {
   RECORD yield();
}
```

3.3 Runtime Efficiency

Our third goal is to control the overhead introduced by DT. To this end, we directly compile DT programs into

[1]TraceSQL also provides function APIs to modify the application stack under special scenarios, as described in Section 5.2.2.

[2]The dynamic instrumentation may still cause heisenbugs, which change or disappear once the debugger is deployed. We attempt to minimize such effect by reducing the memory footprint of the DT system.

TraceSQL Language Overview				
Statement types			**Keywords**	**Comments**
Configuration statements			@fileoutput, @buffersize	Configure the TE environment.
Variable declaration statements			@, INTEGER, LONG, STRUCT, ARRAY	Declare TraceSQL variables. Each statement must start with @.
Tracepoint declaration statements	General		TRACE, FROM, EXECUTE, WHERE, EXIT	The basic format is TRACE {… } FROM {… } EXECUTE {…} WHERE {….}.
	Tracepoint type declarations	Function tracepoints	BEFORE_PROLOGUE, AFTER_PROLOGUE, BEFORE_EPILOGUE, AFTER_EPILOGUE, regular expressions	BEFORE_PROLOGUE inserts the probe before the prologue section of a function generated by GCC, AFTER_PROLOGUE the opposite, etc.
		Statement tracepoints	N/A	N/A
		Virtual tracepoints	PERIOD, FOR, REPEAT	Set up the virtual timer period and the number of times it will be triggered.
	Condition predicates		AND, OR , NOT	Define complicated predicates.
	Tracepoint actions	Generic	RECORD, LOAD, INTO, AS	Operate files in the LiteFS file system.
		On memory variables	READ, SET	Read and write memory variables.
		On invoking functions	INVOKE	Invoke a function.
		On thread operations	TERMINATE, BREAK	Control threads.
Built-in functions	Stack operations		pushinteger(), popinteger(), getsp()	Push and pop integers to and from the stack, and read the stack pointer.
	Number operations		getRand()	Return a random number.
	Memory operations		readMem(), writeMem()	Read and write memory locations directly.

Table 1: Overview of TraceSQL Language Keywords

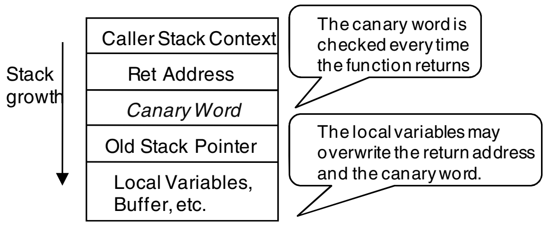

Stack growth
Caller Stack Context
Ret Address
Canary Word
Old Stack Pointer
Local Variables, Buffer, etc.
The canary word is checked every time the function returns
The local variables may overwrite the return address and the canary word.

Figure 2: The Use of Canary Words in StackGuard

application-specific TEs, instead of interpreting them by a common engine running on sensor nodes. Different TraceSQL scripts generate different TEs, and only their binary images are installed. This approach significantly reduces their memory footprint, and reduces the amount of data transferred in the deployment phase of the DT system.

4. THE TRACESQL LANGUAGE

In this section, we systematically describe the syntax of TraceSQL. Table 1 shows an overview of TraceSQL keywords. We first present a complete TraceSQL example, followed by describing its syntax details.

4.1 TraceSQL Program Example

In this example, we present the TraceSQL implementation of the StackGuard tool. As described in Section 2, StackGuard addresses the stack corruption problem through canary words, as illustrated in Figure 2. This problem is usually not caused by security attacks in sensor networks. Instead, it is more common for a node to crash for accidentally overwriting the return address of its functions, causing a function to return to unpredictable addresses. Therefore, using StackGuard is very much needed. While StackGuard is implemented as a patch for GCC, to our knowledge, it does not support AVR-GCC, and hence does not work with sensor networks.

Our implementation of StackGuard is illustrated in Table 2. This example demonstrates the use of integers, tracepoints declarations, and their associated actions. It consists of two probes for each instrumented function, which, in this example, is the `crashNode()` function from the file `app.c`. One probe is located before the prologue, and the other after the epilogue. When a procedure is invoked, the first triggered tracepoint pushes a randomly generated canary word onto the stack. Just before this function returns, the second tracepoint is triggered, which checks if the canary word is still intact. If not, an error is detected and the thread is suspended.

```
1    INTEGER @canarynumber = getRand();
2    INTEGER @testnumber;
3    TRACE crashNode() BEFORE_PROLOGUE
4    FROM app.c EXECUTE
5    {
6       push_integer(@canarynumber);
7    }
8    TRACE crashNode() AFTER_EPILOGUE
9    FROM app.c EXECUTE
10   {
11      @testnumber = pop_integer();
12      IF (@testnumber != @canarynumber) {
13         BREAK;
14      }
15   }
```

Table 2: TraceSQL Example for StackGuard

4.2 Program Structure Overview

Formally, a TraceSQL program consists of three types of statements: configuration statements, variable declaration statements, and tracepoint declaration statements. The example in Table 2 shows the latter two types of statements.

Configuration statements: These statements specify TE parameters. For example, if a program writes to a file, it specifies the name of the file with a configuration statement. Starting with a pre-defined @, the following sample shows how to set the file output and the internal buffer size.

```
@fileoutput = “trace.log”;
@buffersize = 64;
```

Variable declaration statements: These statements specify the types and names of variables. Integers are most commonly used, as shown in lines 1 and 2 of the StackGuard example (Table 2).

Tracepoint declaration statements: Tracepoint declaration statements are the key components of a program.

Each statement consists of three parts, tracepoint addresses, tracepoint actions, and optional condition predicates. One or more tracepoints can be specified within a single statement, and they can also be nested to perform complicated tasks. The declaration part starts with a TRACE keyword, the action part starts with an EXECUTE keyword, and the predicate starts with a WHERE keyword. Hence, a tracepoint declaration looks as follows. Two examples of this declaration are shown in lines 3 and 8 of the StackGuard example (Table 2).

TRACE {...} FROM {...} EXECUTE {...} WHERE {...}

4.3 Tracepoint Declarations

TraceSQL supports three types of tracepoints: function tracepoints, statement tracepoints, and virtual tracepoints.

Function tracepoints: The first type of tracepoints inserts probes into functions. In TraceSQL programs, the probe is addressed by a pair consisting of a function name and a file name. By default, the probe is located before the first instruction generated by the function. Additional keywords, such as BEFORE_PROLOGUE and AFTER_PROLOGUE, can be used to specify the exact address of inserted probes (for these two keywords, they specify that the probe is located before and after the prologue code generated by the GCC compiler).

Function tracepoints can express a variety of events of interest, such as receiving a packet (by adding a probe to the packet receiving function), or reading ADC sensors (by adding a probe to the ADC reading function). To simplify adding a group of tracepoints, the TraceSQL compiler supports regular expressions. For example, the following code inserts probes to all functions starting with device in files with names starting with hardware.

```
INTEGER @counter;
TRACE [device\w*]() FROM [hardware\w*]
EXECUTE {
   @counter++;
   RECORD "Device driver called"+ counter + "\n" ;
}
```

As illustrated in this example, regular expressions make inserting a group of tracepoints much faster than manually instrumenting the source code.

Statement tracepoints: The second type of tracepoints inserts probes into specific source code statements. It is defined by a pair consisting of a line number and a file name. An example is shown as follows:

```
TRACE 236 FROM app.c EXECUTE {
    RECORD "Line 236 reached";
}
```

In this example, the probe is inserted before the first instruction generated by code line 236 in the user file app.c. Compared to function tracepoints, statement tracepoints provide more flexible positioning.

Virtual tracepoints: The third type of tracepoints does not insert explicit probes. Instead, they are triggered by timers, either once or periodically, to perform specific actions. In the following example, we show how to record the value of a counter 100 times at a period of 100 seconds. Observe that TRACE is followed by a variable counter, instead of a function. No confusion will be introduced because unlike functions, variables are not followed with parentheses.

```
TRACE counter FROM app.c PERIOD 100s FOR 100
EXECUTE { RECORD counter; }
```

4.4 Condition Predicates

The TraceSQL compiler also supports condition predicates with the WHERE keyword. A condition predicate can be constructed by logical operators such as AND, OR, and NOT. To illustrate their uses, we present an example that records all context switches triggered by radio transmissions. Here, the msend mutex variable in the library file radio.c is a flag that shows whether or not there is ongoing radio operation (if there is, the msend mutex is locked). We use the READ keyword to access memory variable values.

```
TRACE yield() FROM syscall.c EXECUTE {
   RECORD yield() ;
}
WHERE {
   READ msend->lock FROM radio.c == 1
}
```

4.5 Tracepoint Actions

In this section, we describe the associated actions when tracepoints are reached. The RECORD keyword used in previous examples represents a logging action. Formally, TraceSQL supports the following three action categories: actions on declared variables, on memory variables, and on function invocations.

First, a tracepoint action may operate on declared variables. Consider that a user application does not keep track of the number of received packets and hence provides no visibility on network activity statistics. Without modifying the source code, we can write a TraceSQL program to gather such information by declaring additional variables. In the following example, we define such a variable, numOfPacketsReceived, for this purpose. This variable is shared by two tracepoints, but no race conditions will be introduced because action statements are implicitly protected by atomic operators.

```
INTEGER @numOfPacketsReceived = 0;
TRACE packetreceived() FROM user.c EXECUTE {
   @numOfPacketsReceived ++ ;
}
TRACE PERIOD 100s FOR REPEAT EXECUTE {
   RECORD @numOfPacketsReceived;
   @numOfPacketsReceived = 0;
}
```

Second, a tracepoint action may directly read/write RAM variables using keywords READ and SET. The values of these variables can be combined with other statements to perform complex actions. This has been illustrated in previous examples.

Finally, a tracepoint action may directly invoke functions provided by the operating system or user applications. In TraceSQL, these actions start with the keyword INVOKE. The following example shows how to blink the LED every time a packet is received. The greentogglefunction() is located in the syscall.c file that toggles the green LED.

```
TRACE packetreceived() FROM user.c EXECUTE {
   INVOKE greentogglefunction() FROM syscall.c;
}
```

TraceSQL also provides several built-in functions, as illustrated in Table 1, including stack operations, number operations, and memory operations. The detailed descriptions are skipped because they are self-explanatory.

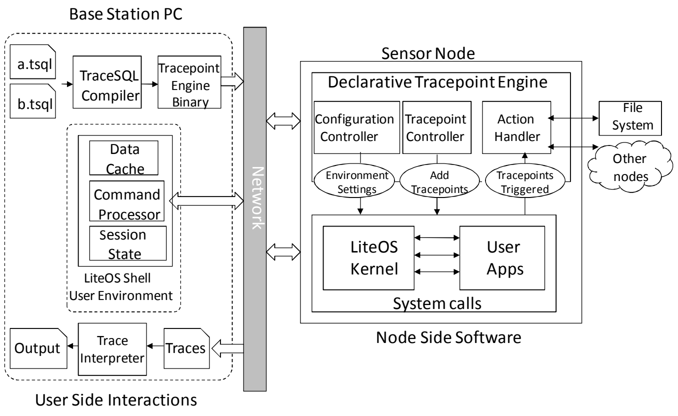


Figure 3: System Architecture for Declarative Tracepoints on LiteOS

5. IMPLEMENTATION

We implement DT on the LiteOS operating system for the MicaZ platform running the Atmega128 processor. In this section, we first describe the execution model of LiteOS, then we describe the details of tracepoint instrumentation.

5.1 Execution Model of LiteOS

LiteOS supports separate compilation of the kernel and user applications. Initially, only the LiteOS kernel is installed. It then loads more applications into memory according to users' needs. The kernel and user applications are bridged by system calls, a special type of function pointers. Multiple user applications can be loaded simultaneously, running as individual threads. As AVR processors provide separate program space (flash) from the data space (RAM), each thread in LiteOS owns a non-overlapping chunk of both the flash memory and the RAM.

Figure 3 shows how we implemented the DT system in the LiteOS environment. Each tracepoint engine (TE) is implemented as a stand-alone thread in LiteOS. It is loaded by the user using the interactive shell, where individual threads can be started or stopped separately. Hence, multiple TEs can be loaded concurrently or sequentially. When one TE terminates, it restores the applications it has instrumented to their original state, so that the normal execution of user applications will not be interrupted.

A tracepoint engine is compiled by the TraceSQL compiler. Implemented in Python, the compiler translates user programs from TraceSQL into C with equivalent semantics. To optimize memory variable operations, the compiler must be provided with memory addresses of the applications. Such information is provided by the `.lss` files generated from user applications.

As output, the TraceSQL compiler generates translated C programs that are ready to be compiled. We use GCC 4.1.1 in our experiments. If the `RECORD` command is used, the compiler also generates a customized trace interpreter for translating the collected traces into readable output.

Note that, our implementation description assumes that traces can be easily obtained at the end of the experiments using the file copy command supported by the LiteOS shell. This assumption is operating-system-dependent. On other operating systems where this command is not available, DT must also implement its own trace retrieval module. Such discussions are out of the scope of this paper.

5.2 The Tracepoint Engine (TE)

This section describes the details of the Tracepoint Engine. As shown in Figure 3, it consists of three modules: the configuration controller for setting up the environment, the tracepoint controller for instrumenting tracepoints, and the action handler for triggered actions.

The first module, the configuration controller, sets up the operating environment. In this part, it opens files for trace output in the LiteOS file system (LiteFS). LiteFS is a hierarchial file system that provides Unix-like operations on the external flash available on the motes. For MicaZ, the size of the flash is 512K bytes. To reduce writing operations, TE also keeps an internal buffer to temporarily store traces. The remaining two modules are more complicated, and we describe them separately.

5.2.1 Tracepoint Instrumentation

Tracepoint instrumentation is based on *dynamic instrumentation*, a technique that modifies application binary code at runtime. On MicaZ, DT relies on the binary rewriting capability of the Atmega128 processor to modify application binaries. It inserts branch instructions into specified locations of original user application binaries as tracepoint portals. The displaced user application instructions are mirrored in the action handlers. Note that, the displaced instructions cannot contain destinations of other branch instructions in the original application. Otherwise, the correct execution of the original application can no longer be enforced (another branch instruction could jump to the middle of the tracepoint portal, causing unexpected errors). This limitation is typically not a problem, because the probability that one instruction being the destination of another branch instruction is measured to be very low. In our benchmark application used later, for instance, the probability is 0.76% (38 instructions are branch destinations in around 5000 instructions). Such destinations of branch instructions are identified and avoided by the TraceSQL compiler.

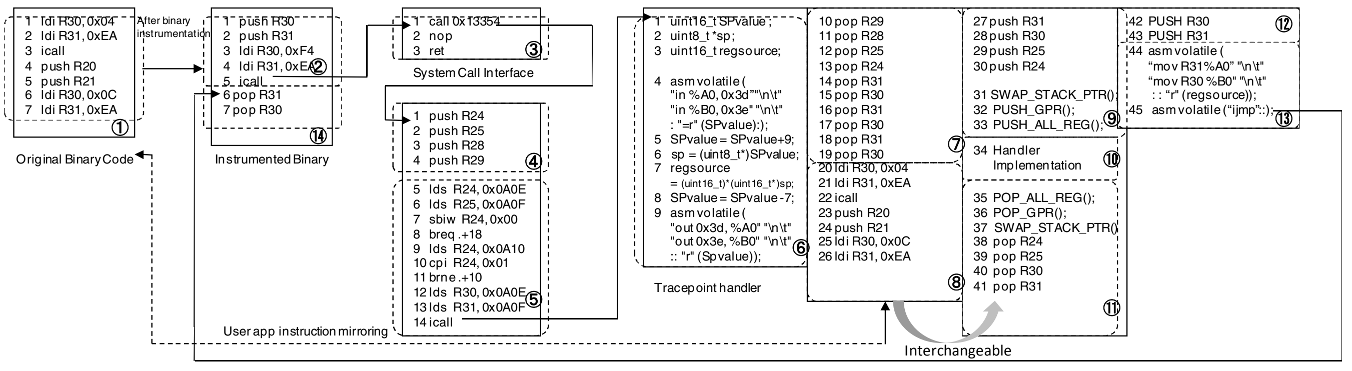
We next describe the implementation of different types of tracepoints.

Function and Statement Tracepoints

Figure 4 shows an example of inserting a tracepoint into a sequence of instructions. For clarity, we organize the whole instrumented code into 14 parts. They are described in detail as follows.

Part 1 is the original binary code (in assembly form), taken from the `greenToggle()` function. After binary instrumentation, these instructions are replaced with a tracepoint portal, shown in part 2. These new instructions first save registers in part 2, then branch to the system call gate in part 3. Parts 3 through 5 check if a tracepoint handler is present, and lead to its handler. Part 6 through part 13 are the core parts of the tracepoint handler, where mixed C and assembly code is used to explain its logic.

At the beginning of part 6, observe that the TE has executed several branch instructions in parts 2, 3, and 5. Each branch instruction pushes a return address onto the stack. Also, the contents of several registers have been modified after they are saved onto the stack. To enforce the correct semantics of the original program, both the stack contents and the register values have to be restored. Parts 6 and part 7 perform these actions, modifying the stack pointer, restoring registers, and saving the program counter to a variable *regsource*.



```
ldi R30, 0x04
ldi R31, 0xEA
icall
push R20
push R21
ldi R30, 0x0C
ldi R31, 0xEA
```

```
push R30
push R31
ldi R30, 0xF4
ldi R31, 0xEA
icall
pop R31
pop R30
```

```
call 0x13354
nop
ret
```

```
push R24
push R25
push R28
push R29
lds  R24, 0x0A0E
lds  R25, 0x0A0F
sbiw R24, 0x00
breq .+18
lds  R24, 0x0A10
cpi  R24, 0x01
brne .+10
lds  R30, 0x0A0E
lds  R31, 0x0A0F
icall
```

```
uint16_t SPvalue;
uint8_t *sp;
uint16_t regsource;

asm volatile (
  "in %A0, 0x3d""\n\t"
  "in %B0, 0x3e" "\n\t"
  : "=r" (SPvalue):);
SPvalue = SPvalue+9;
sp = (uint8_t*)SPvalue;
regsource
  = (uint16_t)*(uint16_t*)sp;
SPvalue = SPvalue -7;
asm volatile (
  "out 0x3d, %A0" "\n\t"
  "out 0x3e, %B0" "\n\t"
  :: "r" (Spvalue));
```

```
pop R29
pop R28
pop R25
pop R24
pop R31
pop R30
pop R31
pop R30
pop R31
pop R30
ldi R30, 0x04
ldi R31, 0xEA
icall
push R20
push R21
ldi R30, 0x0C
ldi R31, 0xEA
```

```
push R31
push R30
push R25
push R24

SWAP_STACK_PTR();
PUSH_GPR();
PUSH_ALL_REG();
Handler
  Implementation
POP_ALL_REG();
POP_GPR();
SWAP_STACK_PTR()
pop R24
pop R25
pop R30
pop R31
```

```
PUSH R30
PUSH R31
asm volatile (
  "mov R31%A0" "\n\t"
  "mov R30 %B0" "\n\t"
  : : "r" (regsource));
asm volatile ("ijmp"::);
```

Figure 4: Implementation for Dynamic Tracepoint Instrumentation

This variable *regsource* is useful in two ways. First, *regsource* differentiates between tracepoint origins. This is needed because TE uses one single handler for all function and statement tracepoints. To map each origin to the right action, TE relies on the value of *regsource*, which has been set as the program counter before the first *icall* instruction in part 2. Second, *regsource* allows the instruction flow to return to the original after the action handler, as shown in parts 12 and 13.

At the end of part 7, the modified registers and the stack pointer have been restored. The mirrored instructions can therefore be safely executed in part 8.

Parts 9, 10, and 11 perform the handler actions. Here TE optionally uses a macro `SWAP_STACK_PTR()` to switch the stack pointer back and forth between the kernel stack and the thread stack. If the original tracepoint is located in a thread, the macro is activated, as is illustrated here. The reason is that the thread stack of user applications in LiteOS is typically compact, and directly executing the action handler may cause stack overflows, especially if complicated actions are involved. Switching to the kernel stack helps solve this problem.

Note that, even with `SWAP_STACK_PTR()`, TE still pushes more variables onto the stack in part 2 through 6 than the unmodified application. In general, TE requires 12 bytes of extra data space to work well. Therefore, if a thread has consumed almost all of its stack space, stack overflow may occur. In our experiments, however, we did not observe this problem because by default, LiteOS allocates more stack than minimum for user applications.

One note on TE is that the mirrored instructions in part 8 can swap with parts 9, 10, and 11, so that the action can also be performed before the mirrored instructions. This feature is useful for implementing the `BEFORE_PROLOGUE` keyword, where the actions have to be executed *before* the displaced prologue section.

System Call Gate and Virtual Tracepoints

DT allows tracepoints to be positioned in system call gates of LiteOS. The system call gates are a table of function pointers through which applications access kernel functionalities. Each system call gate strictly uses 4 bytes. Because of this limitation, the approach described in Figure 4 cannot be applied, as it requires 7 instructions to be modified at the tracepoint portal.

We follow a different approach for system call instrumentation, based on the observation that the total number of system calls in LiteOS is fewer than 80, and their implementation is known. Here, we implemented an instrumented set

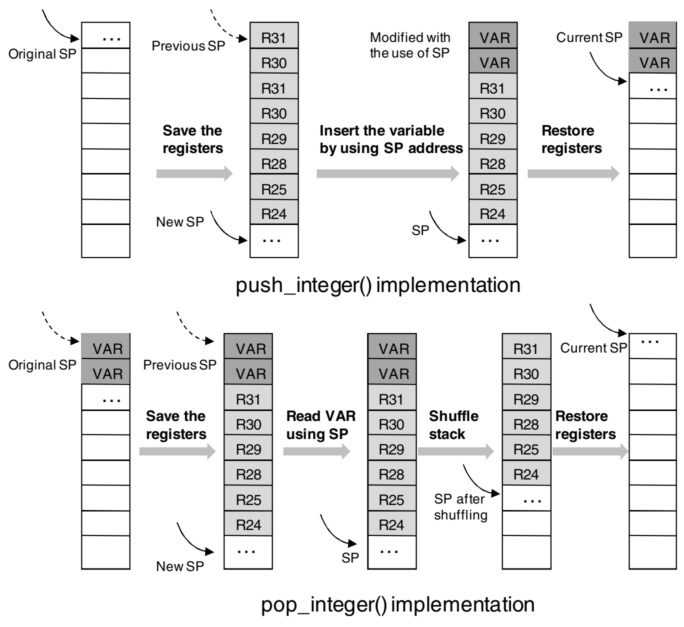


Figure 5: Implementation of Push and Pop Functions

of system call gates that resides entirely in the TE. We then rewrite the original system call gates by only modifying the *call* instructions to point to the instrumented version, so that the system calls are diverted to tracepoint handlers.

A different implementation is needed for virtual tracepoints. Such tracepoints are triggered by timers instead of user applications. Instead of binary instrumentation, for every virtual tracepoint, we set up a callback function that invokes action handlers when the timer fires. The timer can be either one-time or periodic depending on the TraceSQL program.

5.2.2 Action Handlers

We next describe two techniques to implement more complicated actions: stack shuffling and loop compression.

Stack Shuffling Techniques

As illustrated in Table 1, TraceSQL provides two functions: `push_integer()` and `pop_integer()`. Operating directly on the stack, they are unusual in that they explicitly modify the stack content, hence breaking the transparency of the tracepoint handler. So what is the whole point of their existence?

We find two occasions when pushing and popping operations are useful: before prologues and after epilogues. For example, the StackGuard debugging tool checks the return address of functions at these two locations. TraceSQL provides two GCC-dependent keywords, `BEFORE_PROLOGUE` and `AFTER_EPILOGUE` for these needs.

However, supporting the push and pop functions is not trivial. After the stack content is changed, all registers must be restored to their original values. Otherwise, the application logic will be broken. To this end, we developed a technique called *stack shuffling*. The details of stack shuffling are shown in Figure 5.

The implementation of the `push_integer()` function consists of four steps. First, it pushes registers that are used later onto the stack. Notice it pushes R31 and R30 twice, so that there is extra stack space reserved for the integer `VAR` (assumed to be two bytes). Then, the parameter `VAR` is saved to this reserved space with the help of a pointer. Finally, the registers are restored except for the space occupied by `VAR`, then the `push_integer()` operation finishes.

The implementation of the `pop_integer()` function is similar, except that it reads the `VAR` variable rather than saves it. After `VAR` is read, the stack is shuffled by moving stored registers by two bytes while preserving their relative order. Finally, registers are restored and the `pop_integer()` function finishes.

Loop Compression

Loop compression optimizes the `RECORD` operation. In one extreme case, if an event A is logged for 100 times, it should be written as "A for 100 times" instead of writing 100 As repeatedly. The loop compression technique extends this observation by compressing the data stored in the buffer periodically before they are written into an external file. This technique works best for large internal buffers, such as 256 bytes or 512 bytes.

5.2.3 Implementation Notes

In our implementation of tracepoints, we made several tradeoffs. To ensure that the implementation of TE is generic instead of compiler-dependent, we focus on global variable operations in tracepoint actions. The addresses of these variables are obtained by parsing the generated assembly code. For local variables, on the other hand, it is still possible to obtain their values through a pair of `pop` and `push` stack operations. However, TE does not automate this process because in that case, extensive and compiler-dependent analysis on the contents of the stack is required. Such an analysis cannot be ported across compilers easily, in contrast to the way TE handles global variables. For the same reason, TE does not address complexities introduced by compiler-specific optimizations. For instance, it is possible to read out-of-date values for variables that have been temporarily cached by registers. This problem, however, can usually be prevented by carefully selecting tracepoint locations (e.g., at the beginning of functions).

6. EVALUATION OF DT OVERHEAD

In this section, we evaluate the overhead of DT. We focus on three metrics: slowdown in CPU cycles, memory overhead, and flash lifetime. The details of our experiment settings are shown in Table 3, where we carry out all experiments on the LiteOS operating system with MicaZ nodes.

Experiment Platform	
Mote hardware	MicaZ / MIB520 programming board
Compiler	AVR-GCC 4.1.1 WinAVR 20070122
Mote operating system	LiteOS 0.3.2
PC Platform	Windows XP / Cygwin

Experiment Settings 1	
Number of Tracepoints	1 - 15
Actions triggered when tracepoints are reached	None
	Read a memory address
	Write a memory address
	Logging a variable into a file
Size of Buffer	256

Experiment Settings 2	
Number of Tracepoints	Flexible
Addresses of Tracepoints	Kernel
Actions triggered	None
	Logging system call traces into a file
Buffer Size	128, 256, 512 bytes

Table 3: Tracepoint Evaluation Settings

6.1 CPU Slowdown

In DT, new binary code is dynamically weaved into application code at runtime. By analyzing the generated assembly code, we identify that each blank tracepoint (i.e., one without any triggered actions) adds 282 CPU cycles. On MicaZ, this consumes 36 microseconds. The overhead of adding multiple tracepoints is usually tolerable. In one of our later examples where 10 tracepoints are added, each triggered 20 times per second, theoretically, an aggregate overhead of around 0.7% of CPU time is introduced, without considering the effect of their associated actions.

To profile the slowdown impact of tracepoints, we use a simple timer-driven radio message generator as the benchmark application in this evaluation. This application sends out radio messages at a frequency of 20 messages per second. The complete experimental settings are shown in Table 3.

To measure the slowdown effect, we designed a *CPU idleness parameter* (CIP) metric as follows. We modified the LiteOS kernel by adding a loop counter in its main (idle) loop that runs when no tasks or threads are scheduled. When the loop is executed, the counter is increased monotonically while the CPU is idle. Its value is periodically collected through the USART port. The difference in counter readings between two collection instants (i.e., the difference within a period) reflects the amount of idle time of the CPU within one period. The percentage of reduction for this value can be used to estimate CPU overhead.

In the first experiment, we added 1 to 15 tracepoints to the message generator application. We set the size of the file output buffer to 256 bytes. We used the CIP value when no applications are running as the baseline. We then started the application under evaluation, instrumented it with tracepoints, and measured the new CIP value. We plotted the percent of decrease compared to the baseline in Figure 6.

Observe that the message generator consumes around 10% of CIP when no tracepoint is inserted (denoted as 0 on the X axis). As more tracepoints are added, CIP decreases. The average progressive decrease of the CIP counter with each blank tracepoint is 0.0852%, slightly higher than the theoretical 0.072% value based on our earlier analysis. This is expected, because (due to other computations such as the kernel timer) the baseline value of the CIP counter represents less than a 100% idle CPU.

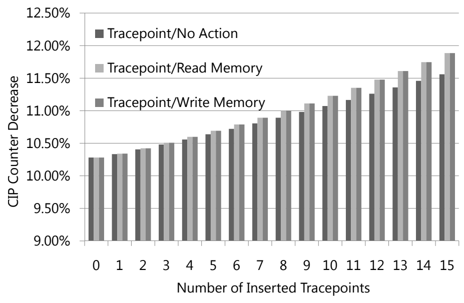


Figure 6: Tracepoints Slowdown by Memory Operations

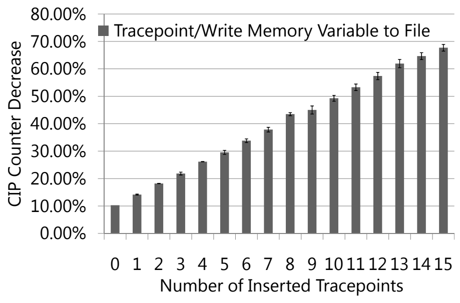


Figure 7: Tracepoints Slowdown by File Operations

To evaluate the slowdown of tracepoints when actions are performed, we measured three cases: when the tracepoint handler reads a 16-bit memory variable, when it writes a 16-bit memory variable, and when it logs a 16-bit variable into an external file. The performance in the former two scenarios is similar to that of blank tracepoints, which is reflected by Figure 6. The performance in the last case is significantly different, and is shown separately in Figure 7. The reason is that file operations consume many more CPU cycles compared to reading/writing memory variables. For file operations, because of the internal buffer, the measured CIP fluctuates. Hence we plot 95% confidence intervals in Figure 7. Observe that in the extreme case, when each of the fifteen tracepoints writes one variable to the file (encoded to 6 bytes), the number of idle CPU cycles drops by almost 70%. For practical applications, however, we rarely log so many variables at once, hence the performance impact is smaller.

In the second experiment, we evaluated the impact of kernel tracepoints. We inserted tracepoints to all system calls, as well as to the radio-sending function in the kernel. The same message generator application is used as the benchmark. As tracepoints are triggered, TE logs either the system call, or the radio sending function. We chose different sizes of file buffers, including 128 bytes, 256 bytes, and 512 bytes. Figure 8 shows the impact of tracepoints with 95% confidence intervals. Observe that, as the buffer size increases, the impact on CPU decreases because file I/O operations are better aggregated. At the end of the experiment, we retrieved the files generated and obtained a complete trace of system calls and radio operations. Table 4 shows one loop in the translated traces, revealing the interactions between the kernel and the user application. Such information provides us with rich details on the behavior of the software stack for debugging purposes. For example, one step in our analysis of the third bug in Section 8.3 is based on such captured details.

6.2 Memory Overhead

In this section, we evaluate the memory overhead of tracepoints. This includes both program flash consumption and RAM consumption. We measured both metrics with the benchmark application. In this experiment, we turned on the `-O3` level of optimization, and enabled full program optimization in the GCC compiler. Regardless of how many tracepoints are inserted, the RAM overhead remains constant, because we only use one tracepoint handler for each TE. The RAM usage for blank tracepoints is measured to

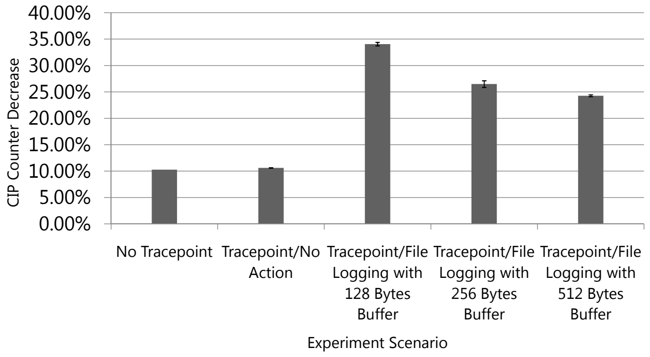


Figure 8: Tracepoints Slowdown with Different Buffer Sizes

be 42 bytes, and for file logging tracepoints 332 bytes, both of which under 10% of the available RAM of MicaZ. Therefore, we consider this overhead to be usually tolerable. For flash usage, we plot our measurement results in Figure 9. As illustrated in this figure, the overhead of compiled code size increases with the number of tracepoints inserted, but is consistently less than 4K bytes, or 3% of the flash available on MicaZ nodes. Therefore, we conclude that the overhead introduced by tracepoints is generally acceptable.

6.3 Flash Lifetime

There are two types of flash lifetime. The first is related to the serial flash used as the file system. Each MicaZ node has 512K bytes of flash memory. Hence, if too much data are written, the flash will be exhausted. In the previous file logging example, each variable is encoded into 6 bytes. Therefore, one tracepoint generates 0.12K bytes of data per second at the chosen frequency. If 15 tracepoints are added, the serial flash will be exhausted in about 280 seconds. In reality, developers should either limit the amount of logging activities, or attach a larger flash to the node [18].

Another flash lifetime parameter is related to the flash memory. On MicaZ, the reprogrammable flash has a lifetime of 10,000 write/erase cycles. Therefore, the total number of debugging rounds for the same application is limited. This is usually not a problem, and in extreme cases, the developer may change memory settings of the compiled programs to balance program flash operations.

7. TRACESQL EXPRESSIVENESS

To highlight the expressiveness of DT and TraceSQL, we show that they can provide the core functionality of several previous debugging techniques. The details of these

```
Partial output from the trace interpreter (one complete loop):
The kernel or app thread has event:\
RADIO_SEND_OPERATION_KERNEL
Thread 2 has event: RESTORE_RADIO_STATE
Thread 2 has event: MUTEX_UNLOCK_FUNCTION
Thread 2 has event: GET_CURRENT_THREAD_ADDRESS
Thread 2 has event: YIELD_FUNCTION
Thread 2 has event: GET_RADIO_MUTEX_ADDRESS
Thread 2 has event: GET_CURRENT_THREAD_ADDRESS
Thread 2 has event: GET_CURRENT_RADIO_INFO_ADDR
Thread 2 has event: GET_CURRENT_THREAD_ADDRESS
Thread 2 has event: GET_CURRENT_THREAD_INDEX
Thread 2 has event: SOCKET_RADIO_SEND_FUNCTION
Thread 2 has event: GET_CURRENT_THREAD_ADDRESS
Thread 2 has event: YIELD_FUNCTION
The kernel or app thread has event:\
RADIO_SEND_OPERATION_KERNEL
```

Table 4: Collected Running Traces

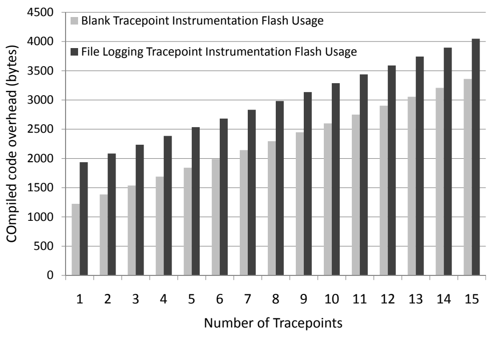


Figure 9: Memory Overhead

techniques are introduced in Section 2. However, these techniques are by no means an exhaustive coverage of the potential of DT/TraceSQL. For EnviroLog, NodeMD, and StackGuard, we implement their core functionality while leaving out less relevant details. For Sympathy, because it is designed in the context of a larger project (ESS), we only briefly outline how it can be expressed.

7.1 EnviroLog

We implemented the core functionality of EnviroLog with TraceSQL. Specifically, we selected the `OscilloscopeRF` application in the EnviroLog 1.0 distribution as the benchmark application (EnviroLog 1.0 contains two examples, the other one is the relatively simple application `Blink`). We implemented and instrumented this application in LiteOS, and our main program is named `OscilloscopeRF.c`. This file invokes the `read_sensor()` function to get readings. It is located on line 61, shown as follows.

```
reading = read_sensor();
```

In EnviroLog, the return value of the function should be captured so that it can be replayed later. We use a statement tracepoint on the following line to implement the record and replay service of EnviroLog as follows.

```
//This is the recording script
@output = "logging.data"
TRACE 62 FROM OscilloscopeRF.c EXECUTE
{
   RECORD reading FROM OscilloscopeRF.c;
}
```

```
//This is the replay script
@input = "logging.data"
INTEGER @loggedreading;
TRACE 62 FROM OscilloscopeRF.c EXECUTE
{
   LOAD INTEGER FROM @input
   INTO @loggedreading;
   SET reading FROM OscilloscopeRF.c
   AS @loggedreading;
}
```

EnviroLog Implementation on NesC/TinyOS		
Metrics	Results (bytes)	Comments
Flash without EnviroLog	11842	For Mica2
RAM without EnviroLog	510	For Mica2
Flash with EnviroLog	27002	For Mica2
RAM with EnviroLog	1319	For Mica2
Increase in Code Size	**15160 (flash) 809 (RAM)**	**Less platform-dependent**
LOC (Lines of Code)	**1**	**Instrumenting the ADC function**

TraceSQL Implementation on LiteOS		
Metrics	Results (bytes)	Comments
Flash of OscilloscopeRF	720	For MicaZ
RAM of OscilloscopeRF	35	For MicaZ
Flash of Logging Service	2014	For MicaZ
RAM of Logging Service	218	For MicaZ
Flash of Replay Service	1992	For MicaZ
RAM of Replay Service	218	For MicaZ
Increase in Code Size	**4006 (flash) 436 (RAM)**	**Less platform-dependent**
LOC	**14**	**Instrumenting the ADC function**

Table 5: EnviroLog Implementation Comparison

Table 5 shows the comparison results of the TraceSQL version of EnviroLog and the original version in terms of compiled code size. Note that, because of the programming paradigm differences between TinyOS (for EnviroLog) and LiteOS (for TraceSQL), the compiled applications have very different sizes. TinyOS compiles applications together with its kernel into a monolithic binary image, whereas LiteOS compiles applications independently of its kernel. Another difference is that EnviroLog supports Mica2. In contrast, TraceSQL currently only supports MicaZ.

We chose two less platform-dependent metrics to make meaningful comparisons, the lines of code written by the user, and the *increase* in the number of code bytes because of EnviroLog. Arguably, these two metrics should be less platform-dependent (because EnviroLog does not directly implement drivers) and less operating-system-dependent (because we are only considering the increase in bytes).

For this particular example, the user of original EnviroLog needs to write one line of code, whereas in TraceSQL, a total of 14 lines of code is needed. Note that the TraceSQL version requires less memory in part because it leverages the file system abstraction provided by LiteOS, while the original EnviroLog implementation has to address EEPROM and serial flash operations at a lower level. Another reason is that the TraceSQL version does not need to implement interactive actions, as they are already provided through the LiteOS

Original Implementation of NodeMD (published in Table 2 of [14])		
Metrics	Results	Comments
Flash	3556 (bytes)	Implemented on Mantis
RAM	302 (bytes)	Implemented on Mantis
LOC	Not available	Not available

TraceSQL Implementation of NodeMD		
Metrics	Results	Comments
Flash	1922 (bytes)	Implemented on LiteOS
RAM	226 (bytes)	Implemented on LiteOS
LOC	22	Incomplete implementation in that no remote retrieval and logging is implemented

Table 6: NodeMD Implementation Comparison

shell. The original EnviroLog implementation, in contrast, provides additional commands such as `START_RECORD` and `START_REPLAY`, thus requiring a larger memory footprint.

7.2 NodeMD

NodeMD detects stack overflows and potential livelocks or deadlocks. The stack overflow problem occurs when the stack collides with the allocated variables (i.e., the `.bss` section). While stack estimation tools exist, they are not always accurate, and hence runtime stack overflow protection is still needed. When a procedure is called, the stack pointer is modified because of three components: the call overhead such as the return address, the passed parameters, and the local data for the function being called. Because the GCC compiler first modifies the stack pointer before using stack memory, NodeMD detects stack overflows by inserting a check code at the start of a function, reading the pre-incremented SP pointer, and comparing this pointer to the stack top (i.e., the end of the `.bss` section). If the SP pointer collides with the latter, a stack overflow is detected.

In our implementation of NodeMD with DT/TraceSQL, we focus on its ability to detect stack overflows and livelock/deadlocks. For its remaining features, its remote retrieval module can be provided by the data copy command in the LiteOS shell. Its application specific fault detection module is based on the *ASSERT* macro, and is supported by the LiteOS programming environment. Therefore, we do not address these two modules in our expressiveness study.

DT follows a runtime modification approach to implement the stack overflow detection of NodeMD, where it inserts a probe immediately before the first line of a function, but after the prologue generated by GCC, so that the stack pointer has been pre-incremented. When the probe is triggered, TE checks whether the SP pointer has reached the stack top. For example, for a function `foo()` from the file `app.c`, we develop the following code.

```
INTEGER @currentsp;
//The stack top for a thread is a known constant
INTEGER @stacktop = ...;
TRACE foo() AFTER_PROLOGUE
FROM app.c EXECUTE
{
   @currentsp = getsp();
   IF @currentsp <= @stacktop
      BREAK;
}
```

The second problem that NodeMD addresses, livelocks and deadlocks, is solved by adding source code checkpoints that registers with the kernel regularly if the thread is alive. NodeMD argues that for most threads, a *timeout* value can be estimated. The thread is expected to reach its checkpoint during each timeout period. Hence, if the thread fails to reach the checkpoint for much longer than *timeout*, the kernel assumes that this thread has reached a deadlock or livelock state.

In TraceSQL, the NodeMD approach to check deadlocks or livelocks can be appropriately implemented using virtual tracepoints. In the following example, TE adds a checkpoint to the `foo()` function in the file `app.c` (which is already running) as shown below. Note that, the `BREAK` keyword can optionally take the name of the thread, assumed to be `app`, as its parameter.

```
INTEGER @checkpointreached;
//The timeout is known for this thread.
LONG @timeout = ...;
TRACE foo() FROM app.c EXECUTE
{
   @checkpointreached = 1;
}
TRACE PERIOD @timeout*3 FOR REPEAT
{
   IF (@checkpointreached == 1)
      @checkpointreached = 0;
   ELSE
      BREAK app;
}
```

Similar to EnviroLog, we use the lines of code written by the user and the memory footprint as metrics to evaluate our implementation of NodeMD with TraceSQL. Because NodeMD source code is currently not publicly available, we refer to the data reported in its original paper. We also use its examples, where two applications are measured. One is `blink_led`, which uses a single thread to periodically toggle an LED, and the other is `FireWxNet`, a multi-tiered monitoring application. We only implemented the first one, and instrumented the `blink_led` application with the TraceSQL version of NodeMD. As illustrated by the comparison results in Table 6, the overhead introduced by TraceSQL is measured to be lower than that of NodeMD. This is probably due to that we only implemented the core features of NodeMD, while leaving out several functionalities that are already implemented in LiteOS.

7.3 StackGuard

We introduced the mechanism of StackGuard in Section 2. The implementation of StackGuard in TraceSQL takes 15 lines of code, as shown in Table 2. The evaluation results of StackGuard are shown in Table 7, where the memory footprint with 1, 5, 10, 20, and 50 functions instrumented is presented. Because the original version of StackGuard does not support the AVR-GCC compiler used for MicaZ, no comparison between the TraceSQL version of StackGuard and the original version is made.

We also designed one improvement over StackGuard without using the canary word. Observe that any return address cannot be zero. Hence, our improved method also checks the return address in each probe for such invalid numbers that will crash the node. This approach turns out to be very useful in one of our case studies to locate a bug in the LiteOS kernel, as shown in Section 8.1.

7.4 Sympathy

Because Sympathy is designed for ESS, we have not implemented its functionality. Instead, we only present a brief overview on how the metric collection subsystem of Sympathy could be expressed with TraceSQL. Table 8 shows the types of metrics that are collected by Sympathy. To

TraceSQL Implementation of StackGuard	
Number of Functions Instrumented	Flash/RAM Usage
1	1178/64
5	2054/64
10	2958/64
20	4952/64
50	10952/64

Table 7: Implementation of StackGuard

Types	Metric Details
Connectivity Metrics	Routing Table Neighbor List
Flow Metrics	Packets Transmitted Packets Received Sink Packets Transmitted Sink Packets Received Sink Last Timestamp
Node Metrics	Node Uptime Bad Packets Received Good Packets Received

Table 8: Sympathy Metrics

demonstrate how they can be implemented in TraceSQL, we summarize their potential implementation in Table 9.

8. CASE STUDIES

One of the most convincing ways to evaluate the effectiveness of a debugging system is to use it to find real bugs. In this section, we present three case studies. In the first two cases, we use the DT to retroactively test existing applications with documented bugs. In the third case, we describe our experiences using DT to debug a routing protocol that we developed for the communication stack of LiteOS.

The motivating question we want to answer in the first two case studies is, could DT have detected any documented old bugs if it were available? We obtained the full documented bug list of the LiteOS operating system from October 2007 to March 2008, when it evolved from version 0.2 to 0.3, and used them to test DT software. Note that, all these bugs had been fixed by the time we tested them with DT. Therefore, we knew the causes of the bug in advance when we did the following tests. This, however, does not prevent us from gaining insight into the strength and limitations of the DT system as a debugging tool.

We focus on a subset of *difficult* bugs that are related to memory. They share the commonality that while the symptom is obvious, it is unclear what variables to watch or where breakpoints should be set. It would be great if the DT system could identify the source of these bugs with less time and effort.

Among the nine *difficult* memory bugs documented (difficulty measured by the time to fix them), we identified two bugs that could have been solved using DT/TraceSQL. This relatively low coverage ratio is primarily caused by the diversity of the bugs, ranging from erroneous pointer dereferences to stack overflows, most of which simply fall outside the scope of DT. For those bugs not caught by DT, they can be found by more conventional methods such as the built-in interactive debugger of LiteOS. For the two bugs that are caught, each took more than one day to solve originally without DT. In contrast, it took less than two hours to develop, debug, and identify the root cause using TraceSQL scripts. This improvement demonstrates the benefits and effectiveness of the DT system.

Metric	TraceSQL Implementation
Routing Table	Read application specific table data structure at runtime with `READ`.
Neighbor List	Read the application specific neighbor list at runtime with `READ`.
Packets Transmitted	Declare a counter for the number of transmitted packets. Add a function tracepoint to the packet send function, and increases the counter every time this tracepoint is reached.
Packets Received	Similar to the previous, except that the function tracepoint should be added to the packet receive function.
Sink Packets Transmitted (the number of sink packets received by the node)	Declare a counter for the number of transmitted sink packets. Add a function tracepoint to the packet receive function, and check if the packet is a sink packet. If it is, increase the counter.
Sink Packets Received	Implemented separately on the sink node, not resource-constrained.
Sink Last Timestamp	Implemented separately on the sink node, not resource-constrained.
Node Uptime	Declare a counter as the uptime. Add a periodic virtual tracepoint, and increase the counter every time the virtual tracepoint is reached. Use the counter as indicator for node uptime. If the node reboots, the counter is reset.
Bad Packets Received	Declare a counter for this metric. Add a statement tracepoint to the line of code where a corrupted packet is received and discarded (e.g., CRC failed). Every time this tracepoint is reached, increase this counter.
Good Packets Received	Similar to the previous one. Add the tracepoint to the statement where the packet passes the CRC check.

Table 9: Expressing Sympathy Metrics with TraceSQL

8.1 Bug I: Node reboot after changing the compiler optimization level

The first bug has the following symptom. When the LiteOS kernel is compiled with `-Os` (optimization for size) instead of `-O0` (no optimization), the node repeatedly reboots itself. Since the kernel invokes many functions in its startup stage, it took a considerable effort to pinpoint the exact buggy function.

To use DT to analyze this bug, our guess is that reboots are usually caused by stack corruptions. Using StackGuard is therefore promising. However, the original version of StackGuard fails to catch this bug. We then tried our improvement that only checks the return address. This time, the bug is caught. The function `lite_switch_to_user_thread`, shown in the following table, is identified as trying to return to a zero address.

```
void __attribute__(( noinline ))\
lite_switch_to_user_thread()
{
  #ifdef PLATFORM_AVR
    PUSH_REG_STATUS();
    PUSH_GPR();
    SWAP_STACK_PTR( old_stack_ptr,\
    current_thread->sp );
    POP_GPR();
    POP_REG_STATUS();
  #endif
  _enable_interrupt();
  return ;
}
```

Technically, this function performs context switches. In this sense, it does not return to its caller. When `-O0` is used, the assembly code generated for this function first pushes registers R28 and R29 into the stack. When `-Os` is used, however, the compiler detects that such two pushes are not necessary, and removes them to generate compact code. However, as shown in lines 7, 8, and 9 of the program, the number of registers to be popped when the context switch occurs is pre-calculated. When two pop instructions are removed, when the context switch occurs for the first time, the stack of the user thread is of the wrong size. Hence, the return instruction at line 11 leads to address 0x00, and reboots the node.

8.2 Bug II: User applications put into MEMORY_CORRUPTED state once executed

The second bug appeared when during one LiteOS kernel update. The symptom was that after this update, any application, once loaded by the kernel, was quickly put into `MEMORY_CORRUPTED` state. Further investigation shows that the kernel puts threads into this state only if the thread appears to be occupying an incorrect chunk of RAM or program flash. This checking procedure is based on reading the thread control block. If the allocation has conflicts with the kernel or other threads, the thread is immediately put into error state. Consequently, the problem is, what caused every thread to have incorrect control block values?

This bug is difficult to solve because the user application appears to be doing nothing wrong. Analysis of the source code does not lead to anything. The bug was later detected with assembly-level analysis.

This bug could have been solved faster with the DT system. In our experiments, we instrumented each function of the user application with a tracepoint that checks whether the thread control block in question has been illegally modified. We used a simple Blink application as the test case, and the instrumented code still triggered the same bug. Hopefully, this approach allows us to localize the problem to the function level so that the search scope is reduced significantly. To accurately pinpoint the function, the tracepoints were inserted after the epilogue section of each function.

With DT deployed, the bug was found immediately. It turned out the control block of threads is modified by the following function, which is intended to put the current thread into sleep mode.

```
1    void sleepThread(int milliseconds) {
2        thread **current_thread;
3        current_thread = getCurrentThread();
4        (*current_thread) ->state = 4;
5        (*current_thread) ->data.sleepstate.sleeptime\
         = milliseconds;
6        yield();
7    }
```

Further investigations reveal that the unintended modification is caused by a mismatch between the thread control block declaration in the kernel and in user applications. During the kernel update, the thread control block in the kernel is updated with additional member variables. On the application side, however, an old, inconsistent version of the thread control block structure is used. When the statement 4 in the above function changes the structure member variable `state` of the control block, it mistakenly modifies the variables that represent the RAM allocation information, because it calculates the offsets incorrectly. Therefore, the thread is blocked and put into the `MEMORY_CORRUPTED` state as soon as the kernel takes over.

8.3 Bug III: Unexpected corruption of communication protocol neighbor table

The third bug is related to our development of routing layer protocols in the LiteOS environment. Here, communication protocols are developed as stand-alone files that are running as individual threads. In our development phase, we attempted to evaluate the performance of multiple protocols by comparing their delivery ratio. However, when nodes switch from the first protocol (geographic forwarding) to the second (logical coordinate based routing, or LCR [5]), the neighbor table of the second protocol sometimes gets corrupted with incorrect values. This symptom is relatively rare, making it hard to pinpoint the exact source of the bug.

We first used the interactive debugger of the LiteOS shell to monitor the neighbor table contents. This approach was not successful, because we can only detect the bug after the neighbor table has been incorrectly modified. Once this happens, the LCR protocol fails to operate correctly. We later decide to automate the bug capture process with TraceSQL. Specifically, we check whether the neighbor table has been corrupted at the end of several functions that we consider might have introduced the bug, by reading neighbor table contents and checking their validness in TraceSQL. To correlate the incorrect modifications with their origins, we also use TraceSQL to capture the traces of system calls, a technique we introduced earlier in Section 6.1.

By analyzing the traces we collected, we identified that the bug symptom is closely correlated with the event of receiving neighbor beacon packets sent by a node running the geographic forwarding protocol. This is normal because nodes do not switch from the first protocol to the second simultaneously. As such a packet arrives, the TE running on the node that has switched to LCR immediately detects that its own neighbor table is incorrectly modified. This insight allows us to pinpoint the exact location of the bug, and solves the problem.

Technically, this bug is introduced by the fact that different communication protocols in LiteOS are differentiated through their port numbers, which are registered by each protocol with the kernel. A callback function is associated with the unique port number of each protocol to perform tasks such as updating the neighbor table. In the failed experiments, after the geographic forwarding protocol process is terminated, its registered callback function is not properly removed as the protocol neglects to de-register itself with the kernel. This problem causes the bug. When a neighbor beacon arrives for the geographic forwarding port, the registered callback function for this port is still invoked. As the two protocols are compiled in such a way that only their RAM allocation overlaps, the binary code of the previous protocol still resides in the program flash, whose execution modifies the RAM locations that are currently occupied by the new protocol. Thus, the neighbor table contents of the latter are incorrectly modified.

9. CONCLUSIONS AND FUTURE WORK

This paper proposes the declarative tracepoint debugging system, the first comprehensive system that allows application-independent and programmable tracepoints to be inserted and removed at application runtime for wireless sensor networks. We demonstrate that its programming language, TraceSQL, is expressive enough to implement the core functionality of a variety of debugging techniques in the literature, such as EnviroLog, NodeMD, Sympathy, and StackGuard. We also demonstrate the effectiveness of the DT system through a series of case studies based on the LiteOS operating system, and demonstrate that it is feasible to use TraceSQL to detect these otherwise difficult bugs.

In future work, we plan to investigate implementing distributed tracepoints, and use the DT system to diagnose new bugs. Such explorations will help us obtain better understanding on the strength and limitations of the DT debugging system.

Acknowledgements

We gratefully acknowledge the anonymous reviewers and our shepherd, Kay Römer, for their insightful comments. This paper is supported in part by NSF grants CNS 05-54759, CNS 06-15318, CNS 06-26342, and CNS 06-26825, and a fellowship from Vodafone Foundation.

10. REFERENCES

[1] Atmel Corporation. Mature AVR JTAG ICE. http://www.atmel.com/dyn/products/tools-card.asp?tool-id=2737.

[2] D. Gay. Design of matchbox, The simple filing system for motes. Available at http://www.tinyos.net/tinyos-1.x/doc/matchbox-design.pdf.

[3] The DTrace Homepage on Sun Microsystems. Website: http://www.sun.com/bigadmin/content/dtrace.

[4] S. Bhatti et al. Mantis OS: An embedded multithreaded operating system for wireless micro sensor platforms. In *ACM/Kluwer Mobile Networks and Applications (MONET), Special Issue on Wireless Sensor Networks*, 2005.

[5] Q. Cao and T. Abdelzaher. Scalable logical coordinates framework for routing in wireless sensor networks. *ACM Transactions on Sensor Networks*, Volume 2, Issue 4, 2006.

[6] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He. The LiteOS operating system: Towards Unix-like abstractions for wireless sensor networks. In *Proceedings of IPSN*, 2008.

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

[8] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, Jan 1998.

[9] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - A lightweight and flexible operating system for tiny networked sensors. In *Emnets-I*, 2004.

[10] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin. EmStar: A software environment for developing and deploying wireless sensor networks. In *USENIX Annual Technical Conference, General Track*, pages 283–296, 2004.

[11] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for network sensors. In *Proceedings of ASPLOS-IX*, 2000.

[12] J. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *ACM SenSys*, November 2004.

[13] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *European Confernece on Object-Oriented Porgramming, volume 1241 of Lecture Notes in Computer Science*, pages 220–242, 1997.

[14] V. Krunic, E. Trumpler, and R. Han. NodeMD: Diagnosing node-level faults in remote wireless sensor systems. In *ACM MobiSys*, pages 43–56, 2007.

[15] P. Levis and D. E. Culler. Mate: A tiny virtual machine for sensor networks. In *Proceedings of ASPLOS*, December 2002.

[16] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire TinyOS applications. In *ACM SenSys*, November 2003.

[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 *IEEE INFOCOM*, 2006.

[18] G. Mathur, P. Desnoyers, D. Ganesan, and P. J. Shenoy. Capsule: An energy-optimized object storage system for memory-constrained sensor devices. In *ACM SenSys*, pages 195–208, 2006.

[19] W. P. McCartney and N. Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In *ACM SenSys*, 2006.

[20] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. In *ACM SenSys*, pages 255–267, 2005.

[21] B. Titzer, D. K. Lee, and J. Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proceedings of IPSN*, pages 477–482, 2005.

[22] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. In *Proceeedings of EWSN*, 2005.

[23] M. Wachs, J. I. Choi, J. W. Lee, K. Srinivasan, Z. Chen, M. Jain, and P. Levis. Visibility: A new metric for protocol design. In *ACM SenSys*, 2007.

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