LIS is More: Improved Diagnostic Logging in Sensor Networks with Log Instrumentation Specifications

Roy Shea
Computer Science Dept.
Univ. of California, Los Angeles
Los Angeles, CA 90095-1594
Email: roy@cs.ucla.edu

Mani Srivastava
Electrical Engineering and
Computer Science Depts.
Univ. of California, Los Angeles
Los Angeles, CA 90095-1594
Email: mbs@ucla.edu

Young Cho
Information Sciences Institute
Univ. of Southern California
Los Angeles, CA 90292-6611
Email: youngcho@isi.edu

***Abstract*—Detailed diagnostic data is a prerequisite for debugging problems and understanding runtime performance in distributed embedded wireless systems. Severe bandwidth limitations, tight timing constraints, and limited program text space hinder the application of standard diagnostic tools within this domain. Our work introduces the Log Instrumentation Specification (LIS) that drives insertion of low overhead logging calls into a system. The LIS language is easy for developers to directly integrate into their daily work flow and, by acting as an intermediary language, facilitates rapid construction of higher level analysis.**

Through microbenchmarks of a complete LIS implementation for the TinyOS operating system, we demonstrate that LIS can comfortably fit onto bottom tier embedded systems. We show how we have used LIS to create a complete monitoring infrastructure for wireless sensor networks that uses features of the LIS language to optimize the size of gathered logs, replicate features of specialized logging infrastructures with minimal effort, and jump start debugging of these systems. Finally, we provide examples of our use of LIS to diagnose specific problems and understand system behavior of a sensor network.

***Keywords*-software engineering; testing and debugging; debugging aids; diagnostics; distributed debugging; sensor networks**

I. Introduction

Innovative research in wireless and embedded sensing systems is enabling larger and more complex wireless sensor network deployments. But these advances in communication, sensing, and processing capabilities of sensing systems are outpacing the development of tools required for developers to understand the faults appearing in deployments and on testbeds. Continued growth of the distributed wireless embedded systems field depends on developing sufficient monitoring to provide diagnostic data required to understand system problems and performance.

Gathering this diagnostic data from wireless embedded systems is very difficult. Physical access to distributed embedded systems is often unavailable requiring diagnostic techniques that do not depend on physical device access and, especially with wireless systems, limiting the diagnostic bandwidth available. Physical coupling between sensor systems and the environment, and time sensitive interactions among members of the distributed network, greatly limits the applicability of interactive debugging solutions that suspends execution of the debugged device. This physical coupling with the environment also limits the utility of simulators that too often fail to capture key subtleties of the physical world in the modeled sensor input streams or radio models. This shortcoming appears time and time again in practice where, after having thoroughly tested a sensor network in simulation, a team must engage in a new round of debugging and development to handle new problems that arise in the actual deployment or large testbed deployment.

Our answer to the need for an in-situ logging framework is the Log Instrumentation Specification (LIS) language that provides developers with the support needed to obtain high quality logs from distributed embedded systems with minimal deployment impact. Performance analysis of LIS shows that it is well suited for the tight program memory, RAM, and bandwidth constraints of the wireless embedded systems domain. We present tools that use LIS to provide a window into wireless embedded systems by exposing detailed runtime traces, monitoring system health, and aiding the debugging of timing sensitive subsystems that are tied to the physical world. The tools presented perform a wide array of monitoring tasks that illustrate the expressiveness of the LIS language, its ability to gather logs in an efficient manner, and its amenability to optimization.

II. LIS System Architecture

The LIS framework provides the complete infrastructure required for developers to easily and concisely expose runtime system state. Figure 1 illustrates the core components of LIS within the context of its work flow: a LIS script, the instrumentation engine, and the runtime `bitlog` library. Developers compose a description of the information they wish to gather by writing a LIS script, such as that in Figure 2, or by using higher level analyses that output LIS as an intermediary language. This script drives instrumentation of the application

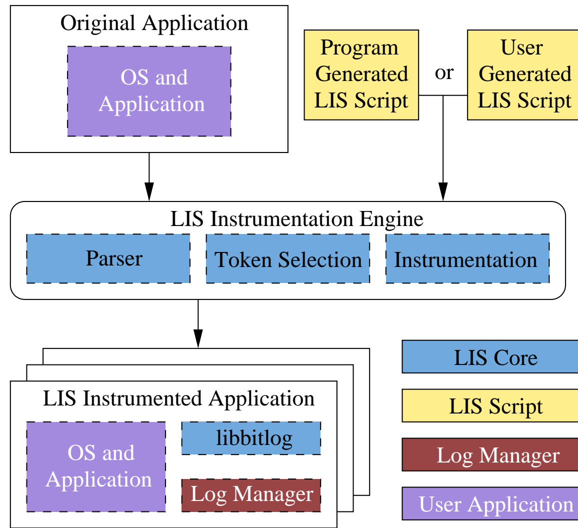

Fig. 1. Architecture of LIS.

```
header read_done global
controlflow read_done local if send_busy
footer read_done point
```

Fig. 2. LIS script used to instrument the `read_done` function.

resulting in an application with calls to a narrow logging interface. Figure 3 shows a function before and after being instrumented by the specification in Figure 2. The logging interface is provided by the standalone `bitlog` library that has been carefully constructed to minimize its resource demands, allowing it to fit comfortably into the resource constrained domain of embedded wireless sensor networks. This section presents the LIS language, the implementation of the core LIS components, and evaluates an instantiation of LIS for TinyOS.

A. LIS Language

The LIS language describes logging tasks. The LIS language provides a core set of logging primitives that balance clarity and portability with expressibility and efficiency. Primitives used by LIS are readily understandable by developers to encourage exploratory writing of LIS scripts. The expressibility of LIS allows higher level analyses performing complex or optimized logging tasks to use LIS as an intermediate language. The narrow LIS interface also encourages a more focused logging mentality, as opposed to the ad-hoc scattering of log statements often associated with manual log instrumentation.

Table I shows the complete LIS grammar. A LIS script is made up of one or more statements describing the information to be logged within a system. Each LIS statement includes three common fields describing the *instrumentation type* that the statement is a member of, a *scoping declaration* from which the logged token identifier(s) resulting from the statement should be drawn, and a *placement specifier* stating the name of a function within which to apply the statement. Some instrumentation types include additional type-specific fields.

The placement specifier and instrumentation type work together to describe where logging statements should be

```
/* Pre-LIS */
void read_done(error_t result, uint16_t data) {
   if (send_busy == TRUE) return;
   /* Rest of function body elided... */
   return;
}

/* Post-LIS */
void read_done(error_t result, uint16_t data) {
   bitlog_write(4, 3); /* Header LIS statement */
   if (send_busy == TRUE) {
      bitlog_write(5, 3); /* Control flow LIS statement */
      bitlog_write(0, 1); /* Footer LIS statement */
      return;
   }
   bitlog_write(6, 3); /* Control flow LIS statement */
   /* Rest of function body elided... */
   bitlog_write(0, 1); /* Footer LIS statement */
   return;
}
```

Fig. 3. Listing of the pre- and post-LIS extended `read_done` function.

TABLE I
LIS GRAMMAR

$Start$	$\rightarrow$	$Statements \mid \epsilon$
$Statements$	$\rightarrow$	$Stmt\ Statements \mid Stmt$
$Stmt$	$\rightarrow$	$Header \mid Footer \mid Call \mid ControlFlow \mid Watch$
$Header$	$\rightarrow$	**header** $Placement\ Scope$
$Footer$	$\rightarrow$	**footer** $Placement\ Scope$
$Call$	$\rightarrow$	**call** $Placement\ Scope\ Target$
$ControlFlow$	$\rightarrow$	**controlflow** $Placement\ Scope\ Flag\ Var$
$Watch$	$\rightarrow$	**watch** $Placement\ Scope\ Var$
$Placement$	$\rightarrow$	F
$Scope$	$\rightarrow$	**global** $\mid$ **local** $\mid$ **point**
$Target$	$\rightarrow$	$F \mid$ **__PTR__**
$Flag$	$\rightarrow$	**if** $\mid$ **switch** $\mid$ **loop** $\mid$ **if − switch** $\mid$ **if − loop**
		$\mid$ **switch − loop** $\mid$ **if − switch − loop**
Var	$\rightarrow$	$\langle$Varible name from program$\rangle \mid$ **__ANY__**
F	$\rightarrow$	$\langle$Function name from program$\rangle$

inserted within an application. A placement specifier is simply a function name that binds the application of a LIS statement to the specified function within a program. The instrumentation type, described in more detail in Table II, specifies where within the function the LIS statement should be applied. Some instrumentation types use additional fields to further limit the application of the LIS statement. For example, the control flow instrumentation type allows specification of the type of control flow statements to consider and a variable name that must be present in the guard for the LIS statement to be applied.

At the heart of LIS are the three scoping declarations that direct the assignment of the token identifier that is logged by the system as a result of a given LIS statement and are summarized in Table III. The runtime uses the minimum number of bits required to log a given token. Explicit name space assignment allows aggressive log size optimization exploiting multiple name spaces. A demonstrative use of name spaces to significantly reduce log sizes is presented in Section III-A.

TABLE II
LIS INSTRUMENTATION TYPES

Instrumentation Type	Description
Header	Write token immediately upon entry into function specified by the placement field.
Footer	Write token immediately before returning from function specified by the placement field.
Call	Write token immediately before calling specified target function from within function specified by the placement field. Each such call uses the same token identifier.
Control Flow	Write a token at the head of each branch reachable from specified control flow statements within the function specified by the placement field. Control flow statements are specified using a flag describing the statement types to track: *if*, *switch*, and / or *loop*. Can be optionally filtered to only log branches guarded by an expression using the named variable. Global and local scoping results in a unique token value for the top of each branch.
Watch Point	Write a token immediately after any assignment to the specified variable within the function specified by the placement field. After writing this token the value of the variable is written to the log. Each watch point generated by a single LIS statement uses the same token identifier.

TABLE III
LIS SCOPING DECLARATIONS

Scope	Description
Global	Token value unique among all LIS tokens.
Local	Token value unique within the host function. Token value may appear in local name spaces for other functions.
Point	Single unique token value.

Important to the design of LIS are the styles of logging that it discourages. LIS discourages logging arbitrary values. Token identifiers are assigned by the instrumentation engine, and the only other data recorded are variable values resulting from watch points. LIS discourages arbitrary placement of logging statements within a system, with logging statements constrained to the placement governed by the five instrumentation types. These design decisions help users focus on a logging task by providing structure and preventing users from descending into the details of manual system instrumentation.

B. LIS Implementation

A LIS implementation requires three components: an instrumentation engine that instruments the application based on a LIS script, a runtime infrastructure providing the logging primitives used by the instrumentation engine, and a log parser.

The LIS instrumentation is implemented using the C intermediary language [1] and we have used this implementation to instrument C for both embedded wireless sensor networks and general systems applications. The instrumentation engine is quite small since the primitives used by LIS are very basic and require minimal program analysis from the instrumentation engine. Core primitives provided by the instrumentation engine are the ability to log tokens in a function header, a function footer, before a function call, at the head of each branch immediately after a control flow decision, and after variable assignment.

We implemented the platform independent `bitlog` library to provide runtime logging support for LIS. The `bitlog_init` function is called once by host system during platform startup to initialize the `bitlog` library. Tokens and variable values are logged by calling `bitlog_write` that takes a 32-bit data value and an 8-bit width, writing the lower width bits of data to a buffer managed by the `bitlog` library. Whenever the log buffer managed by `bitlog` fills, the library calls `bitlog_flush` to flush the buffer out for storage or transportation. Applications can also call `bitlog_flush` to force flushing of the log buffer.

LIS includes a generic log parser that tokenizes logs generated by a wide variety of LIS scripts. This parser does depend on knowing the current scope to decode locally scoped tokens, although LIS scripts can be written that do not generate logs with this information and such scripts require minor extensions to the parser to define how scope can be determined to decode locally scoped tokens. In the case of missing or corrupted log data, most often from incomplete logs caused by dropped log data packets, the parser resynchronizes at a later point in the log data stream and notes how many bits of log data were dropped during the scan. The parser currently prints a textual description of tokenized traces describing the location within the code base generating each token in the log.

C. LIS Evaluation

LIS is carefully designed to minimize impact on the embedded system where it is used, while maximizing the portability to facilitate its use on a wide variety of embedded systems. The core runtime support provided by the `bitlog` library is implemented using standard ANSI C features and is thus easily portable to new platforms that have a C compiler. Overhead from using LIS comes from the static cost of introducing logging statements and the LIS runtime into the instrumented system, and from the runtime costs associated with calling into the logging infrastructure.

We evaluated LIS by integrating it into TinyOS, a commonly used operating system for sensor networks that uses the nesC programming language [2] to compose embedded systems. LIS works directly with C source code, so we added it as an optional compilation stage within the TinyOS build system placed after nesC generates a C representation of the application and before the architecture specific C compiler builds the application. This new stage uses a LIS script specified by an environment variable to drive instrumentation of the application. Developers can write this LIS scripts by hand or use a higher level analysis that generates LIS output.

We implemented a log manager for TinyOS called `LogTap` to handle logs after they have been flushed by the `bitlog` library. Our implementation of `LogTap` can route logs to a sink within the network using the collection tree protocol

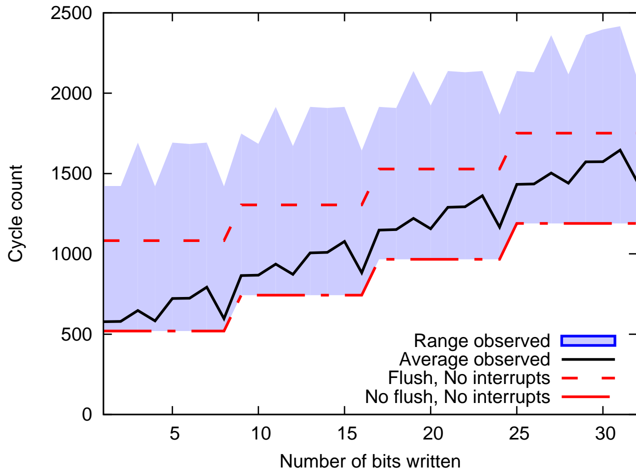

Fig. 4. Range of cycle counts observed from calling `bitlog_write`.

[3] (CTP) or send them directly using the TinyOS `AMSend` interface. The latter option is particularly useful for sending logs out over serial interfaces on testbeds or for broadcasting logs in deployments where CTP is not desired.

Table IV lists the program memory and RAM overheads resulting from adding the `bitlog` library and the `LogTap` TinyOS component on different hardware platforms. system. Also listed in the table is the overhead resulting from adding a single call to the `bitlog_write` function provided by `bitlog` and used by LIS to log tokens.

LIS introduces latency into the systems as a result of the one time call to initialize `bitlog`, calling into the `bitlog` library to log data, and as a result of flushing the log buffer back into the host operating system. We profiled the latency introduced by these calls using the cycle accurate Avrora [4] simulator. Table V lists the run time latency in cycle counts observed on an ATMega128 processor resulting from each call into `bitlog`. The minimum cycle count for an operation is the cost of the call when no external interrupts delay its execution. In practice, interrupts from normal system activity may fire during the handling of these library routines and introduce more latency. We also examined cycle counts resulting from calling into these functions in the TinyOS operating system with a light level of background activity interrupting the functions and used these observations to find the average and maximum cycle counts observed under a typical system load.

Of particular interest to users of LIS are the expected and maximal costs of calling `bitlog_write`, since this is the logging call inserted by LIS into user code. Figure 4 shows a detailed view of the latency resulting from `bitlog_write` calls of various data widths. The minimum number of cycles observed is the result of calling the function when no interrupts occur. This minimum number of cycles is a step function directly related to the number of bits written and primarily results from the bit shift operations used to handle 8-bits chunks of data `bitlog_write` function. The shaded region shows the range of cycle counts observed. Of note is that the average number of cycles observed remains close to the minimum for all data widths. Also included on the plot is the latency resulting from a call to `bitlog_write` that causes a call to `bitlog_flush` when no interrupts occur.

The source of worst case observed latency results from `bitlog_flush` copying a buffer, 26 bytes in the `AMSend` version `bitlog` library, from the logging library into the TinyOS `LogTap` component. When `LogTap` uses CTP the log buffer is reduced to 16 bytes to make room for routing layer headers and the flush delay is slightly reduced. For systems with an extreme latency intolerance the impact of flushing a buffer during a call to `bitlog_write` can be almost completely eliminated by integrating the `bitlog` library into TinyOS and using buffer swapping. This increase in performance would come at the cost of decreased portability of the `bitlog` library.

A final source of runtime overhead results from the transmission (or storage) of a log after it is flushed from the `bitlog` library. Transmission of buffers in the TinyOS implementation is handled by posting a send request to `LogTap` after a log flush. The posted request is handled only after the user code triggering the flush has returned.

We feel that these overheads are quite reasonable for the benefits provided by logging. While logging in time critical systems requires an awareness of these overheads, we have found LIS a valuable asset to understanding problems both in and out of timing sensitive systems. We have even used LIS to diagnose timing problems within the CC2420 radio stack, as described in Section III-C.

III. Applications of LIS

Key monitoring challenges in the wireless embedded network domain include exposing detailed traces of program execution to help developers understand their systems, remotely monitoring overall system health, and debugging time sensitive subsystems that include intimate couplings to the physical world. LIS provides a foundation upon which these task can be easily and efficiently accomplished.

A. *Region of Interest Call Trace Monitoring Using LIS*

Development of LIS was motivated by the authors' frustrations with debugging wireless and embedded sensing systems and first came to life as a custom call tracing utility for sensor network operating systems. Logs of call traces through a user specified Region Of Interest (ROI) provide the information needed to learn more about a code base. Simply knowing the precise sequence of calls made within a suspect subsystem is often enough to isolate a problem or jump start debugging using more invasive tools, such as a software debugger or JTAG.

1) Global Identifier Logging: Many utilities focus on log analysis, rather than optimizing log collection, and thus create logs in a direct manner by assigning each logged event a unique identifier from a global token name space. Examples of this approach are seen throughout the wireless sensor network community such as in the SNTS project [5] and subsequent

TABLE IV
STATIC OVERHEAD OF LIS ON DIFFERENT PLATFORMS

	Mica2 (ATMega128)		MicaZ (ATMega128)		TelosB (MSP430)	
	Program Memory	**RAM**	**Program Memory**	**RAM**	**Program Memory**	**RAM**
TinyOS Radio Stack	7178	137	9264	210	8456	229
TinyOS Collection Tree Protocol	9692	1230	10284	1360	11126	1295
LogTap (using CTP)	1384	303	1412	351	2228	353
LogTap (using radio broadcast)	108	113	74	128	388	131
Bitlog Library	386	43	386	43	362	43
Call to `bitlog_write`	14	0	14	0	12	0

TABLE V
CYCLE COUNTS OF LIS LOGGING ACTIONS

Action	Avg.	Min.	Max.
`bitlog_init`	129	129	129
`bitlog_write` (0 bits)	311	297	463
`bitlog_write` (1 - 8 bits)	653	520	1692
`bitlog_write` (9 - 16 bits)	939	743	1915
`bitlog_write` (17 - 24 bits)	1223	966	2138
`bitlog_write` (25 - 32 bits)	1507	1189	2417
`bitlog_flush_data`	593	562	733

work with Dustminer [6] where each event of interest is assigned a unique identifier from a global token name space.

This ease of constructing logs from global identifiers motivates *global identifier logging*, a technique where each function within the ROI logs a unique identifier when it is called. By examining the stream of identifiers logged by the system, the order of calls made through the ROI at run time is available. The addition of a marker logged when a function from the ROI returns allows the nesting of function calls to also be reconstructed. Global identifier logging is trivial to implement on top of LIS. The ROI is constructed by filtering the list of functions within the program using regular expressions supplied by the user. For each function in the ROI, a globally scoped header LIS statement and a point scoped footer LIS statement is generated for that function.

2) Local Call Logging: An alternate to global identifier logging is *local call logging* [7]. Local call logging reduces the average width of logged tokens by breaking the large name space used by global identifier logging into multiple smaller locally scoped name spaces. This often results in a significant reduction in the resulting log size. This is functionally accomplished by having callers, rather than callees, log a local caller scoped identifier describing the function being called. This does require that the scope of a token in the log can be determined when the log is being parsed. We have written an analysis on top of LIS to handle this added complexity and, given an ROI, generate a LIS script performing local call logging. Assuming a user-specified ROI, the local call logging analysis classifies functions within a program as:

- *External functions* that are not members of the ROI and through which calls are not logged.
- *Entry functions* that are within the ROI and either called by at least one external function or act as entry points to the program (typically the `main` function).
- *Body functions* that are within the ROI and not an entry function.

An initial analysis stage generates a list of all caller and target pairs. For each entry function, a globally scoped header LIS statement is generated. For each function in the ROI, a point scoped footer LIS statement is generated. Finally, for each body ROI function called from a function (note that this caller must also be within the ROI since the target is a body function), a locally scoped LIS entry targeting calls to the target from the caller is generated.

3) Evaluation of ROI Logging Approaches: We evaluated our ROI call trace logging implementations using two TelosB sensor motes running the TinyOS operating system. The TelosB mote uses an MSP430 microcontroller and a CC2420 radio. Each ROI analysis generates a LIS script that is passed to the TinyOS build system to create an instrumented application. The `LogTap` component routes logs out of the network using the collection tree protocol.

We explored gathering call traces from 13 small ROIs that each encompass one TinyOS subsystem and range in size from 2 to 59 functions. To stress our ability to gather logs we continued by combining multiple subsystems into large ROIs that include up to 238 functions. While the input for each of the 13 small ROI is a single word naming the TinyOS subsystem of interest, on average the global identifier and local call logging analyses respectively generate 37 and 42 lines of LIS code. This highlights one of the benefits of LIS – the ease with which it can be used by higher level analyses, such as the ROI tracing, that provide a concise interface to a complex logging task.

Figure 5 presents the run time log bandwidth resulting from instrumenting ROIs of various sizes using global identifier logging and local call logging to implement call tracing. Log bandwidth is calculated using only the 16 byte log payload of the packet and does not count headers used by implementation specific protocols such as the collection tree routing protocol. We use the log sequence numbers to allow us to account for the bits consumed by dropped packets, an event occasionally observed from bursty events overrunning the log buffers and from the naturally lossy radio channels.

Local call tracing consumes less bandwidth for every ROI. For the small single subsystem ROIs, these savings are not significant. The savings become significant for large ROIs, where the local call logging scheme significantly reduced log bandwidth. This is also reflected in packet drop rates. Drop rates were very low for the small ROIs and averaged well

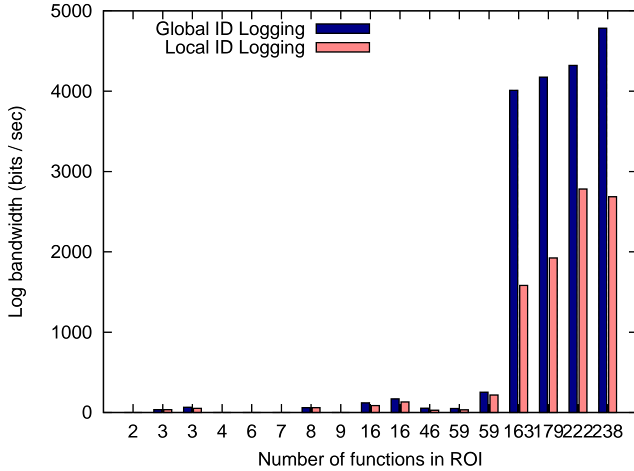

Fig. 5. Log bandwidth resulting from ROIs of various sizes.

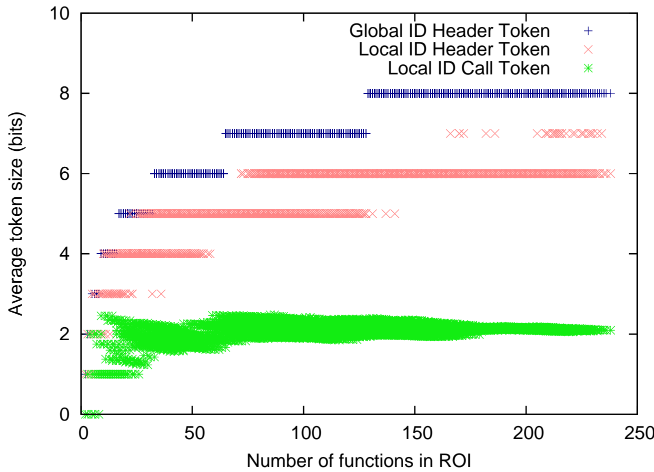

Fig. 6. Token sizes resulting from ROIs of various sizes.

under 1% for both ROI tracing methods. Drop rates for local call logging in ROIs with more than 150 functions peaked at 6.1%, but ranged from 10% to 17% for global identifier logging.

The intuition behind these savings is made apparent by Figure 6. For each of the 8192 ROI possible from combinations of the 13 subsystems examined, the figure plots the average token sizes used by global identifier and local call logging against the size of the ROI. By dividing tokens into entry and body name spaces, local call logging reduces the average token size. Further, the average body token length stabilizes to just over 2 bits as the ROI increases rather than suffering from the logarithmic growth experienced by entry tokens that must be unique. This savings is particularly important for large ROI that begin to stress the system.

Inserting logging into programs does increase the required program memory. For the single subsystem ROI this increase was between 5% and 10%. For the large ROI the increase ranged from 30% to 40%. However, the program memory increase resulting from inserting logging statements never varied by more than 3% between global identifier and local call logging.

```
watch CtpForwardingEngineP.0.sendTask.runTask \
      global dest
watch CtpRoutingEngineP.0.routingTableUpdateEntry \
      global CtpRoutingEngineP.0.routingTableActive
watch CtpRoutingEngineP.0.routingTableEvict \
      global CtpRoutingEngineP.0.routingTableActive
controlflow CC2420ReceiveP.RXFIFO.readDone \
      global 1 buf
```

Fig. 7. LIS script to monitor the network health of devices using the CC2420 radio and CTP for data collection.

The logs collected by the ROI analysis are passed to the LIS parser that outputs per-device call traces. The parser depends on knowing the current scope to decode locally scoped tokens, and this information is always present in complete global identifier and local call tracking logs. Even when more than 50% of the log packets from a given node are lost (a rate significantly higher than that observed in our ROI tests) the parser has been able to extract meaningful call traces, although the missing data causes the traces to be heavily segmented.

LIS facilitates implementing high level logging tasks and exploring logging optimizations. Both global identifier logging and local call logging were easy to implement on top of the framework provided by LIS. Token scoping features of LIS are exploited by the local call logging approach to create an ROI tracing framework significantly more efficient than the commonly used global identifier logging approach.

B. *System Health Monitoring with LIS*

Systems such as Sympathy [8] and LiveNet [9] monitor the overall network health of a deployment. LIS can be used to provide similar health monitoring without requiring manual instrumentation of the code base. Figure 7 shows the complete body of the CC2420 CTP routing health script for TinyOS. For each node in the network, this script provides a stream of data that describes the currently selected next hop, the number of downstream neighbors, and the distribution of CRC failures. On a small four node testbed this script resulted in per-node averages of 0.37 (direct neighbor to the root with no children) to 0.84 (node forwarding data for other nodes) bytes of log data each second.

LIS encourages developers to write general purpose scripts that implement complete logging tasks. Such a script can vary in complexity ranging from the ROI analysis that examines the underlying application to optimize the logging overhead, to direct implementation of a required task such as the network health monitoring presented in this section.

C. *Problem Solving with LIS*

A wonderful feature of LIS its ability to rapidly answer correctness and performance questions about a system during development with minimal deviation from the standard work flow. The ability to answer these questions using only a few seconds of scripting allows work to remain focused on development and implementation. Table VI presents examples

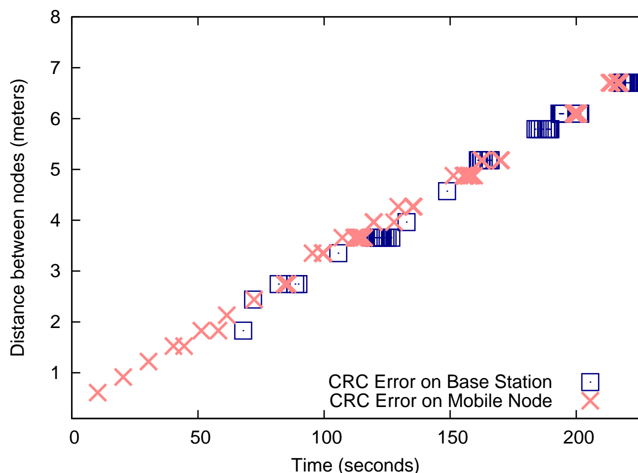

Fig. 8. Instances of CRC errors observed between two TelosB nodes as one moves progressively further away. Precise time of drop is approximated using time stamp from log packet arrival.

of the types of questions we pose to LIS that are discussed in more detail below.

1) What is the ratio and distribution of packets being dropped by each node due to CRC failure?: Many communication drivers include a CRC check to detect bit errors in packets. The one line LIS script in the first row of Table VI monitors the path taken by the TinyOS CC2420 radio driver after checking the CRC value of a packet. Figure 8 shows this data for two TelosB motes with their CC2420 radios configured to use a transmit power of -25 dBm and running `RadioCountToLeds`, an application that broadcasts a counter four times a second, instrumented to monitor CRC checks. Both nodes were placed on the floor of our lab with the base station node remaining in a fixed location and the second node being moved 30.5 centimeters further away every 10 seconds. The base station made 767 CRC checks of which 9.38% failed, compared to the mobile node that made 725 CRC checks of which 9.24% failed.

Figure 8 reveals that the base station experienced a non-negligible rate of CRC failures regardless of transmission distance. A particularly high number of CRC failures appears around 4 meters of node separation (perhaps due to two metal filing cabinets the mobile node passed between at this point) even though the range of the radios with this power is closer to 6 meters where CRC failures peak a second time.

Using the default LIS parser to generate logs describing the branch identifier taken after checking the CRC, we were quickly able to explore CRC errors within our network.

2) Why won't my routing tree form?: Understanding a new embedded code base is a daunting task, especially with research systems that may have little or no documentation. In addition to gathering call traces for a new system as discussed in Section III-A, detailed traces of the control flow taken through complex application code can reveal a great deal about the normal, or abnormal, workings of a function. For example the second entry of Table VI shows a two line LIS script that tracks control flow through the main routing procedure used within CTP.

Table VII shows the normal sequence of events, and our annotations of what the branch does, generated by LIS on non-root nodes sending data over CTP without any problems. When first running LIS with applications using CTP we observed that routing trees never formed, but were able to collect logs using the broadcast runtime for LIS. These logs revealed the sequence of branches for non-root nodes was not that of Table VII, but a shorter sequence occurring when a valid parent is not found. This insight helped us quickly isolate a bug that incorrectly handled integral type promotion of the 16-bit unsigned short type, causing the check for a valid parent to always fail. LIS exposed the runtime decisions made by CTP, allowing us to focus our subsequent debugging on a single function one line long, which was particularly important given the subtle nature of the bug.

3) Where have all the time stamps gone?: Some bugs are nearly impossible to diagnose without the type of low overhead logging that is provided by LIS. One such class of bugs are those of a time sensitive nature where use of interactive debugging techniques to observe intermediate state can violate program correctness properties – such as causing a protocol timeout or suspending execution resulting in a missed event generated by an external system – and thus mask the actual problem which we are trying to observe. A second such class of bugs are those tied to particular hardware devices or physical phenomenon that are abstracted away by simulators, and thus unable to be examined in a simulated environment. We recently used LIS to diagnose a problem spanning both of these classes.

After enabling a resource accounting framework in TinyOS, we observed instability in the packet time stamping performed by the CC2420 radio driver. Time stamps were frequently missing from data packets, causing poor time synchronization performance. Interactive debugging tools such as JTAG simulation were ill suited for this bug since suspension of execution within the radio driver could cause timing violations within the CC2420 radio driver and because external devices (the other side of the radio link) would not be similarly suspended. Simulation of this problem was difficult since the simulation environment needed span many layers including the problematic TinyOS program running on an MSP430 processor trigging the bug, low layer details of the CC2420, and the interaction between wireless devices. We thus turned to diagnostic logs that can be easily produced by LIS.

The ROI tracing analysis provided by LIS was used to examine call traces through the radio driver. The one line ROI specification in the third row of Table VI was passed to the LIS ROI analysis to track execution through the `CC2420TransmitP` TinyOS component. Traces were collected from radio drivers with and without resource accounting enabled, where time synchronization fails and succeeds respectively. Comparing these traces immediately revealed a key difference in the call sequence.

Traces for the radio driver with resource accounting

TABLE VI
SHORT LIS SCRIPTS

Monitoring Task	LIS Script
Monitor distribution of packets with CRC errors.	`controlflow CC2420ReceiveP.RXFIFO.readDone global if buf`

Trace detailed flow that determines if and how an outgoing data packet is routed within CTP.	`header CtpForwardingEngineP.0.sendTask.runTask point`
	`controlflow CtpForwardingEngineP.0.sendTask.runTask local \`
	`if-switch-loop __ANY__`

ROI specification to monitor suspect TinyOS CC2420 subsystem using the LIS ROI analysis.	`CC2420TransmitP` (An ROI specification used to generate a LIS script)

TABLE VII
NORMAL CTP SEND SEQUENCE FOR NON-ROOT NODES

LIS Log Token	Comments
`<--`	Enter `runTask`
`BID: 1`	CTP is not busy
`BID: 3`	Send queue is nonempty
`BID: 32`	Node is not root so must route message
`BID: 33`	Found a route to the root
`BID: 4`	Prepare to send data
`BID: 9`	Neighbor is not congested
`BID: 16`	Message has not already been sent
`BID: 19`	Node is not the root so must route message
`BID: 21`	Found current path quality metric
`BID: 23`	Not congested
`BID: 24`	Succeeded in sending message
`BID: 25`	Note that the client sent a packet
`<--`	Enter `runTask`
`BID: 1`	CTP is not busy
`BID: 2`	Send queue is empty

enabled revealed the handler function for the CC2420 start of frame delimiter (SFD) calling: `CC2420Receive.sfd`, the accounting framework, and then `CC2420Receive.sfd_dropped`. This stood in sharp contrast to traces without accounting enabled where the SFD handler function called neither the accounting framework (not surprising since the framework was disabled) nor `CC2420Receive.sfd_dropped`.

This insight narrowed the problem down to a section of code about 20 lines long. Manual inspection of that code revealed that delay introduced by resource accounting caused a radio event to not be promptly handled, resulting in an incorrectly conservative block of code invalidating time stamps in the message.

LIS suited this task because its compact static footprint allowed it to fit onto the MSP430 processor and, more importantly, because the runtime latency introduced by LIS was small enough that it did not violate tight timing constraints within the CC2420 driver.

IV. RELATED WORKS

Extraction of diagnostic data from modern desktop and cluster systems has been studied for many years leading to a mature selection of debuggers, loggers, and log processing frameworks. Representative debuggers include GDB [10], DBX [11], and the Microsoft Visual Studio Debugger [12]. These debuggers provide detailed insight into the runtime state of an application and heavily utilize features such as conditional break points and stepping through execution. Many application domains, such as kernel debugging and performance monitoring, react poorly to suspended execution resulting from interactive debuggers and turn towards logging, such as that provided by the Linux Tracing Toolkit [13] and Event Tracing for Windows [14]. The sheer volume of the resulting logs, especially in distributed systems where log features are correlated across multiple devices and multiple log types, requires a dedicated monitoring infrastructure to processes log data. Magpie [15], the work of Cohen et. al. [16], and Pip [17] are good examples of research filling this need and show promise in extracting significant features from the extensive logs spanning multiple software and hardware systems. Even with tools to process the gathered logs, the underlying premise to "log everything all the time" overwhelms the limited resources on embedded systems and limits their direct applicability in the wireless distributed embedded systems domain.

Embedded processors often lack the internal hardware support used by a debugger. Debugging in these embedded systems can be provided using in-circuit emulators (ICE) and in-circuit debuggers (ICD) [18], [19]. ICE and ICD provide detailed views into the operation of an embedded program by emulating the embedded processor or controlling the embedded processor through a common interface such as JTAG. ICE and ICD techniques typically require costly external hardware and often require physical access to the embedded system, this makes them most effective for pre-deployment testing of individual components from the distributed systems. When faced with the complete distributed system, all of these techniques are susceptible to timing problems resulting from suspending or significantly slowing execution of a subset of devices. Often LIS can replace these heavier weight approaches by providing enough insight to help developers solve their problems without full ICE or ICD.

Distributed embedded systems development also uses simulation to provided insight into runtime state. Simulators vary widely in their focus and may provide a convenient means to run high level application logic [20], precisely model the underlying hardware [4], or model physical phenomenon such as the radio channel [21]. While each of these points caters towards solving a specific class of problem and are important for initial system development, many problems only appear in actual deployments where alternate tools such as LIS must be applied.

Wireless sensor networks are a particularly challenging subset of the distributed embedded systems domain due to their

limited communication bandwidth (often less than 1 Mbps), extensive interaction between devices, and close couplings between individual devices and the physical world. Much of the monitoring and debugging work within this domain fits into one of three categories: passive external watchers, on-device state loggers, and tools acting on the monitored system.

A commonly proposed technique collects logs using a secondary network of "sniffer" nodes. LiveNet [9] uses sniffed data to learn about properties including network routing paths and hot spots in the network, PAD [22] monitors sniffed data to violations of user specified assertion, and the sensor network troubleshoot tool kit (SNTS) [5] uses frequency analysis to highlight interesting features from sniffed logs. These logging systems work well along side LIS, providing additional depth to the logs LIS collects.

LIS specializes in logging runtime state, a feature that other tools can reuse. Sympathy [8] is a focused tool that gathers logs containing network health metrics of deployed devices at a back end server where they are analyzed to diagnose common network faults. Dustminer [6] combines on node logging with an extension of the analysis from SNTS to help isolate logged event sequences that are likely the root cause of an observed problem. LIS could be used as a replacement logging layer by these types of tools. For example, the token identifier sequences generated by LIS, such as those listed in Table VII, are precisely the type of input that tools like Dustminer [6] act on.

The sensor network management system [23] also logs runtime state, but does so through a free form `printf` interface. In contrast to this type of logging framework, LIS pulls the logging specification out of the code base to keep a clean separation between application code and logging code.

NodeMD [24], interface contracts [25], and Harbor [26] span the boundary between logging and interacting with a device. These tools actively reason about – rather than simply log – runtime state while watching for correctness violations. Other frameworks allow the value of data to not only be observed, but to be set. Marionette [27] provides an interface to remotely query and set arbitrary state, while EnviroLog [28] records low frequency events for later replay of an application component. Declarative frameworks, including the declarative trace points used in TraceSQL [29] and the proposed Wringer [30] (built upon DSN [31]) debugging framework, provide an intriguing mix between the active fault monitoring tools and the more passive logging infrastructures. Clairvoyant [32] provides a complete software debugging environment for wireless embedded systems. While these tools allow nearly arbitrary monitoring and interaction with a system, they are accompanied by significantly more overhead and complexity. As these tools mature and become readily available for the wireless distributed embedded domain, they will fit into an important space for full featured monitoring applications that must interact with the system.

Table VIII categorizes some of the works most closely related to LIS. Note that some of the sizes listed in Table VIII are only point samples of a variable sized system or include extra subsystems as described below:

TABLE VIII
CLASSES AND STATIC OVERHEAD OF MONITORING SYSTEMS

Class	Tool	Program Memory	RAM	Measured Platform
Watchers	SNTS	Uses external hardware		
	LiveNet	Uses external hardware		
	PDA	Uses external hardware		
Loggers	Sympathy	1558	47	TinyOS 1.x / Mica2.
	LIS	1798	394	TinyOS 2.1 / MicaZ.
	Dustminer	14670	830	TinyOS 2.0 / MicaZ.
	NodeMD	3556	302	Mantios OS / MicaZ.
Interactive	EnviroLog	15160	809	TinyOS 1.x / Mica2
	Marionette	1000	140	TinyOS 1.x / TelosB.
	TraceSQL	4006	436	LitOS / MicaZ.
	Clairvoyant	32384	1027	TinyOS 1.x / Mica2.

- Core LIS system listed above. LIS adds about 14 bytes for each logging statement.
- Dustminer size includes the `BlockRead` and `BlockWrite` components used for flash reading and writing.
- NodeMD is variable size, with the above size for the basic `blink_led` program.
- Core Marionette system listed above. Marionette adds about 100 bytes for each RPC stub.
- TraceSQL is a widely varying size based on the tasks it is performing. The above listing is for a reimplementation of EnviorLog using TraceSQL.

Despite ongoing research, most of the tools discussed have yet to find widespread use within the wireless sensor network domain. More commonly observed is the humble, but ubiquitous, "LED debugging". LED debugging provides developers with a log, often maintained in the head of an observer carefully counting LED blinks or frantically scratched onto a scrap of paper, while allowing execution of the system to continue. LED debugging provides a direct visual queue of system state changes that is readily understood by system developers. The next most popular debugging framework after blinking LEDs may be `printf`, which is frequently asked about on both the TinyOS and Avrora user mailing lists. The structured logging provided by LIS combines with its ease of use to make it a great alternative to littering code with blinking LEDs and calls to `printf`.

V. CONCLUSIONS

LIS provides a flexible interface upon which many important monitoring tasks targeted for the distributed wireless embedded systems domain can be built. The ability to understand runtime program behavior using the region of interest analysis built on top of LIS often provides the insight necessary to resolve program correctness and performance questions where other tools fall short. The minimal overhead of LIS allows its use on extremely resource limited microprocessors frequently used in wireless and embedded sensing systems, and its low runtime latency allows LIS to examine delay intolerant segments of code. The simple core language encourages developers to use LIS in their daily work, rather than depending

on ad-hoc and error prone “blinking LED” styles of system diagnosis.

Additional work can help LIS provide a more complete logging framework. LIS scripts can be crafted that create logs that are difficult or impossible to unambiguously parse, and the current infrastructure provides no support to warn users when this is the case. The instrumentation engine does not currently support pointer analysis or referencing into complex data structures, limiting the expressibility of LIS statements. Both of these limitations can be mediated by future expansions to the LIS infrastructure using established compiler and program analysis techniques.

LIS focuses on providing exception logging support, but this is only one aspect of a complete monitoring and debugging framework. LIS is a passive system that creates logs of program state and so it does not natively support revising program state, extending program functionality, or logging arbitrary data at arbitrary times. These design decisions make LIS a concise and compact language that is very portable, amenable to optimization from external analysis, and easy for developers to pickup and use. Existing simulation methods, application-specific monitoring infrastructures, and heavier weight tools that interact with the monitored system provide specialized features beyond LIS to create complete monitoring and debugging coverage of distributed embedded systems.

For the wide range of tasks we encounter in our daily development for and research into wireless embedded sensor networks, we find that LIS provides quick and effective logging support. The complete LIS framework and infrastructure for integration into the TinyOS build system is available online at http://nesl.ee.ucla.edu/projects/lis.

Acknowledgment

This work is funded in part by the National Science Foundation under award # CCF-0820061, and by the Center for Embedded Networked Sensing. Any opinions, findings and conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reect the views of the listed funding agencies.

References

[1] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, “CIL: Intermediate language and tools for analysis and transformation of C programs,” in *CC*. Springer, 2002.
[2] “The nesc language: A holistic approach to networked embedded systems,” in *PLDI*, 2003.
[3] R. Fonseca, O. Gnawali, K. Jamieson, S. Kim, P. Levis, and A. Woo, *TEP123: The Collection Tree Protocol (CTP)*, TinyOS 2.0.2 Documentation, 2006.
[4] B. L. Titzer, D. K. Lee, and J. Palsberg, “Avrora: scalable sensor network simulation with precise timing,” in *IPSN*. IEEE Press, 2005.
[5] M. M. H. Khan, L. Luo, C. Huang, and T. F. Abdelzaher, “Snts: Sensor network troubleshooting suite,” in *DCSS*. Springer, 2007.
[6] M. Khan, , H. Le, H. Ahmadi, T. Abdelzaher, , and J. Han, “Dustminer: Troubleshooting interactive complexity bugs in sensor networks,” in *SenSys*. ACM, 2008.
[7] R. Shea, Y. Cho, and M. Srivastava, “Compressing trace logs for monitoring and debugging of embedded networked systems,” in *SenSys* poster session, 2008.
[8] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin, “Sympathy for the sensor network debugger,” in *SenSys*. ACM, 2005.
[9] B.-R. Chen, G. Peterson, G. Mainland, and M. Welsh, “Livenet: Using passive monitoring to reconstruct sensor network dynamics,” in *DCOSS*. Springer-Verlag, 2008.
[10] “Gdb: The gnu project debugger,” GDB Developers. [Online]. Available: http://www.gnu.org/software/gdb/
[11] *Solaris Application Developers Guide*. Prentice Hall, 1997.
[12] *Visual Studio User Reference*, Microsoft Corporation, 2008.
[13] K. Yaghmour and M. R. Dagenais, “Measuring and characterizing system behavior using kernel-level event logging,” in *USENIX*. USENIX, 2000.
[14] *Event Tracing for Windows*, Microsoft Corporation, 2008.
[15] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, “Magpie: Online modelling and performance-aware systems,” in *HotOS*. USENIX, 2003.
[16] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, “Capturing, indexing, clustering, and retrieving system history,” *SIGOPS*, 2005.
[17] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, “Pip: detecting the unexpected in distributed systems,” in *NSDI*. USENIX, 2006.
[18] *HPUSB, USB, HPPCI and MSP430 Emulators Users Guide*, 2007.
[19] *AVR JTAG ICE: User Gudie*, 2001.
[20] P. Levis, N. Lee, M. Welsh, and D. Culler, “Tossim: accurate and scalable simulation of entire tinyos applications,” in *SenSys*. ACM, 2003.
[21] *QualNet User Manual v 4.5*, 2008.
[22] K. Römer, “Passive distributed assertions for sensor networks,” in *DCOSS*. Springer-Verlag, 2008.
[23] G. Tolle and D. Culler, “Design of an application-cooperative management system for wireless sensor networks,” in *EWSN*. IEEE, 2005.
[24] V. Krunic, E. Trumpler, and R. Han, “Nodemd: diagnosing node-level faults in remote wireless sensor systems,” in *MobiSys*. ACM, 2007.
[25] W. Archer, P. Levis, and J. Regehr, “Interface contracts for tinyos,” in *IPSN*. ACM, 2007.
[26] R. Kumar, E. Kohler, and M. Srivastava, “Harbor: software-based memory protection for sensor nodes,” in *IPSN*. ACM, 2007.
[27] 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 *IPSN*. ACM, 2006.
[28] 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,” *INFOCOM*, 2006.
[29] Q. Cao, T. Abdelzaher, J. Stankovic, K. Whitehouse, and L. Luo, “Declarative tracepoints: A programmable and application independent debugging system for wireless sensor networks,” in *SenSys*. ACM, 2008.
[30] A. Tavakoli, “Wringer: A debugging and monitoring framework for wireless sensor networks,” in *SenSys Doctoral Colloquium*. ACM, 2007.
[31] D. Chu, L. Popa, A. Tavakoli, J. M. Hellerstein, P. Levis, S. Shenker, and I. Stoica, “The design and implementation of a declarative sensor network system,” in *SenSys*. ACM, 2007.
[32] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse, “Clairvoyant: a comprehensive source-level debugger for wireless sensor networks,” in *SenSys*. ACM, 2007.