# Scoped Identifiers for Efficient Bit Aligned Logging

Roy Shea
Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90095-1594
Email: rshea@ucla.edu

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

Young Cho
Information Sciences Institute
University 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 wireless embedded systems. Severe bandwidth limitations, tight timing constraints, and limited program text space hinder the application of standard diagnostic tools within this domain. This work introduces the Log Instrumentation Specification (LIS), which provides a high level logging interface to developers and is able to create extremely compact diagnostic logs. LIS uses a token scoping technique to aggressively compact identifiers that are packed into bit aligned log buffers.**

**LIS is evaluated in the context of recording call traces within a network of wireless sensor nodes. Our evaluation shows that logs generated using LIS require less than 50% of the bandwidth utilized by alternate logging mechanisms. Through microbenchmarking of a complete LIS implementation for the TinyOS operating system, we demonstrate that LIS can comfortably fit onto low-end embedded systems. By significantly reducing log bandwidth, LIS enables extraction of a more complete picture of runtime behavior from distributed wireless embedded systems.**

## 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 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 monitoring techniques that provide the diagnostic data needed to understand runtime system behavior.

Gathering this diagnostic data from wireless embedded systems is very difficult. Physical access to distributed embedded systems is often not available, requiring diagnostic techniques that do not depend on physical device access. Physical coupling of sensor systems to the environment, and time sensitive interactions among members of the distributed network, greatly limits the applicability of interactive debugging solutions that suspend execution of the debugged device. This physical coupling with the environment also limits the utility of simulators, which too often fail to capture key subtleties of the physical world in the modeled sensor input streams and radio models. These limitations motivate use of logging frameworks that gather system state at runtime to help end users understand and diagnose observed behavior within a deployed system.

The most basic form of logging is what we term "`printf` logging", in which a developer calls a logging routine to record diagnostic messages and key program state. These descriptive logs are displayed on a console, written to stable storage, or transferred out of the network. Even when the `printf` function is not used, this style of logging appears time and time again.

A problem long recognized in the wireless embedded systems community is the tension between collecting rich logs and the limited system resources for storage or log transfer. Early sensor network research [1] observed this tension and took an important first step in minimizing log bandwidth by encoding each verbose ASCII string using a shorter unique identifier. When examining such a log outside of the network, a dictionary replaces identifiers with the corresponding original log messages.

An obvious next step, embraced and evaluated in this research, is using bit alignment in logs rather than the more common byte alignment. This shift allows us to reap the full benefits of coding techniques that assign highly optimized sub-byte and variable width tokens.

Receiving the full benefits of bit aligned logging requires better logging interfaces that create opportunities for using extremely small token identifiers. This paper introduces the Log Instrumentation Specification (LIS). LIS is a language used to write simple high level scripts that describe logging tasks. Tokens logged by LIS belong to various scopes. These scopes partition the logged set of tokens into small name spaces, facilitating extremely compact encoding of the tokens.

Our research strives to significantly reduce the bandwidth required to collect logs from wireless embedded systems. To this end we:

- Model the potential gains of bit aligned logging over byte aligned logging to highlight the need for extremely small identifiers;
- Demonstrate the improvement in log bandwidth utilization provided by bit aligning identifiers created using tokens from scoped name spaces;
- Introduce LIS, an implementation of these ideas that fits comfortably on distributed wireless and embedded systems.

## II. LOG INSTRUMENTATION SPECIFICATIONS

Log instrumentation specifications provide the infrastructure required for developers to easily and concisely expose runtime system state. Figure 1 illustrates the complete LIS work flow. An end user begins with a high level *LIS script*, such as that at the top of Figure 2, describing a logging task. In the

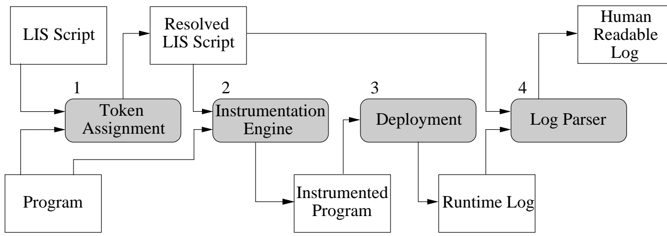


Fig. 1. Complete LIS work flow.

```
/* External LIS Script */
header read_done global
controlflow read_done local if send_busy
footer read_done point

/* 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. 2. A LIS script, the original version of the `read_done` function, and the resulting instrumented version of the `read_done` function.

first stage, the LIS script and program to be instrumented are analyzed to derive the runtime tokens that will be logged and to assign each token a compact identifier honoring the scoping requested in the LIS script. The output of the first stage is a *resolved LIS script* containing the explicit token identifier assignments. Automated instrumentation of the program is driven by this resolved LIS script in the second stage of the work flow. For example, Figure 2 shows relevant portions of the `read_done` function before and after the LIS script is applied to it. The instrumented program can then be deployed to generate runtime logs from one or more devices. In the final stage of the LIS work flow, the collected logs are parsed into a human readable description of the deployed system's behavior. The rest of this section introduces the LIS language and describes the token scoping capabilities provided by LIS that enable compact identifier assignments to tokens.

Developers compose a description of the information they wish to log by writing a LIS script or by using a higher level analysis that outputs LIS as an intermediary language. The LIS language describes logging tasks using a core set of primitives that balance clarity and portability with expressibility and efficiency. LIS primitives are designed to be intuitive, so that they are readily understandable by developers and 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.

TABLE I
LIS GRAMMAR

$$
\begin{aligned}
\textit{Start} &\rightarrow \textit{Statements} \mid \epsilon \\
\textit{Statements} &\rightarrow \textit{Stmt Statements} \mid \textit{Stmt} \\
\textit{Stmt} &\rightarrow \textit{Header} \mid \textit{Footer} \mid \textit{Call} \mid \textit{ControlFlow} \\
&\quad \mid \textit{Watch} \mid \textit{Label} \\
\textit{Header} &\rightarrow \textbf{header}\ \textit{FuncName Scope} \\
\textit{Footer} &\rightarrow \textbf{footer}\ \textit{FuncName Scope} \\
\textit{Call} &\rightarrow \textbf{call}\ \textit{FuncName Scope Target} \\
\textit{ControlFlow} &\rightarrow \textbf{controlflow}\ \textit{FuncName Scope Flag Var} \\
\textit{Watch} &\rightarrow \textbf{watch}\ \textit{FuncName Scope Var} \\
\textit{Label} &\rightarrow \textbf{label}\ \textit{FuncName Scope Var ProgLabel} \\
\textit{Scope} &\rightarrow \textbf{global} \mid \textbf{local} \mid \textbf{point} \\
\textit{Target} &\rightarrow \textit{FuncName} \mid \textbf{__PTR__} \\
\textit{Flag} &\rightarrow \textbf{if} \mid \textbf{switch} \mid \textbf{loop} \mid \textbf{if}-\textbf{switch} \mid \textbf{if}-\textbf{loop} \\
&\quad \mid \textbf{switch}-\textbf{loop} \mid \textbf{if}-\textbf{switch}-\textbf{loop} \\
\textit{ProgLabel} &\rightarrow \langle \text{Label from program} \rangle \\
\textit{Var} &\rightarrow \langle \text{Variable name from program} \rangle \mid \textbf{__ANY__} \\
\textit{FuncName} &\rightarrow \langle \text{Function name from program} \rangle
\end{aligned}
$$

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. 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.

LIS provides six instrumentation types to direct logging within a function:

- *Header statements* log a token upon entry to a function;
- *Footer statements* log a token when a function returns;
- *Call statements* log a token immediately before calling a specified target function;
- *Control flow statements* record the branch taken at particular control flow points within a function;
- *Watch statements* record updates to a specific variable;
- *Label statements* provide for logging of arbitrary data at any point within a function.

At the heart of LIS are the three scoping declarations that direct the assignment of the token identifier(s) logged by the system as a result of a given LIS statement. Local scoping creates multiple small token name spaces that facilitate small bitwidth identifiers. Local tokens are assigned identifiers unique within the host function, although the identifier used for a locally scoped token may also appear in the local name spaces of other functions. Global tokens are each assigned

a value unique among all identifiers. A small set of global tokens are required to provide the log parser with contextual information to correctly parse otherwise ambiguous local identifiers. The point token is a single small unique identifier that can be used by multiple different LIS statements when explicit differentiation between the statements is not needed.

The fine resolution scoping provided by LIS contrasts traditional logging techniques that implicitly use a single global name space to derive token identifiers. A single global name space results in larger token identifiers that, as described in Section III, fail to enjoy the benefits of bit aligned logging. The use of a single global name space is understandable when considering the complexity of manually managing global and local token identifier assignments throughout a program without the automation provided by LIS.

## III. Evaluation

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 software. LIS acts directly on C source code, so we added instrumentation via LIS as an optional compilation stage within the TinyOS build system. This support is 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 direct instrumentation of the application. We implemented a generic logging library called `bitlog` to manage bit aligned logs, and a log manager for TinyOS called `LogTap` to handle logs after they have been flushed by the `bitlog` library. Our implementation of `LogTap` routes logs to a sink within the network using the collection tree protocol [3] (CTP) or sends them directly over a serial link.

The LIS instrumentation engine is implemented using the C Intermediary Language [4]. We have used LIS 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.

Many different techniques could be used to encode tokens within a name space. The evaluated version of LIS uses a simple fixed bitwidth coding scheme to assign each token in a name space a unique identifier. Name spaces containing different numbers of tokens can result in identifiers of differing lengths. An additional one or two bits (depending on scope type) are prepended to LIS generated tokens to enforce separation of the global, local, and point scope classes.

LIS deployments are free to use alternate coding techniques that can provide additional savings in log bandwidth. For example, we have had great success using Huffman coding to assign token identifiers within a name space when knowledge of the approximate runtime token frequencies is available. The key property of the coding scheme used in this evaluation, and true for many coding schemes, is that smaller token name spaces result in the use of identifiers that (on average) contain fewer bits.

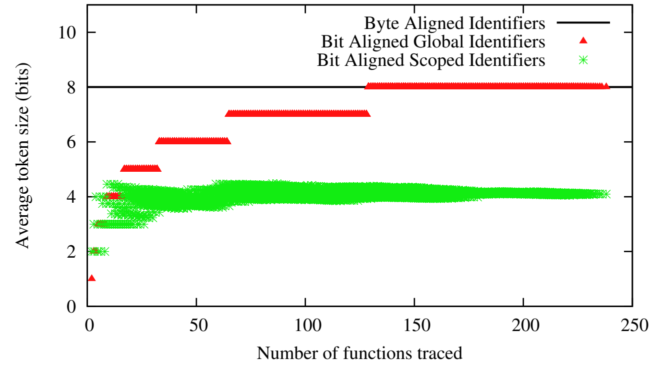


Fig. 3. Average identifier length required for token sets of various sizes. Bit aligned identifiers using scoped token name spaces stabilize to an average length around 4 bits. This tends to out perform the logarithmic growth resulting from bit aligned identifiers assigned to the same set of tokens in a single global name space.

### A. Theoretical Savings from Bit Aligned Logging

The goal of this research is to reduce the bandwidth required to accomplish requested logging tasks. We begin by examining the potential savings that bit aligned logging can provide over a byte aligned system. Let bit_size describe the number of bits needed to encode an identifier $i$ and let byte_size be the number of bits required to encode a byte aligned version of $i$:

$$\text{byte_size}(i) = \left\lceil \frac{\text{bit_size}(i)}{8} \right\rceil * 8 \ .$$

Savings from logging a bit aligned identifier $i$ is bounded by the number of additional bits used by byte alignment:

$$\text{savings}(i) = \text{byte_size}(i) - \text{bit_size}(i) \leq 7 \ ,$$

and the savings relative to the byte aligned size approaches zero as the bit size of $i$ increases:

$$\lim_{i \to \infty} \frac{\text{savings}(i)}{\text{byte_size}(i)} = 0 \ .$$

Consequently, the majority of tokens must be encoded with extremely short identifiers to realize significant benefits from bit aligned logging.

### B. Observed Bandwidth Reductions Provided by LIS

The feasibility of using short identifiers in real systems is evaluated by calculating the average token size used to capture call traces through one or more subsystems within TinyOS. We performed this evaluation using both byte aligned identifiers and bit aligned identifiers from a single name space, and bit aligned identifiers from multiple scoped name spaces. Call traces are captured by logging identifiers to record calls into functions of interest and by logging the subsequent returns from functions of interest. The single name space approach assigns each called function a unique identifier. Local scoping primarily uses identifiers unique to each local caller, as proposed in [5]. Figure 3 shows the average identifier length assigned to tokens used to capture traces from each of 8191 combinations of 13 subsystems within TinyOS for each of

TABLE II
RUNTIME IDENTIFIER SIZE

| Logging Technique | Average Size (bits) | |
|---|---|---|
| | < 60 tokens | > 150 tokens |
| Byte Aligned | 8.0 | 8.0 |
| Bit Aligned Global Scoping | 3.5 | 8.0 |
| Bit Aligned Local Scoping | 2.8 ± 0.5 | 3.7 ± 0.2 |

TABLE III
STATIC OVERHEAD OF THE LIS IMPLEMENTATION ON MICAZ MOTES

| System Component | Program Memory | RAM |
|---|---|---|
| TinyOS Radio Stack | 9264 | 210 |
| TinyOS Collection Tree Protocol | 10284 | 1360 |
| LogTap (using CTP) | 1412 | 351 |
| LogTap (using radio broadcast) | 74 | 128 |
| Bitlog Library | 290 | 24 |
| Call to `bitlog_write` | 14 | 0 |

the three token assignment techniques. Scoping significantly reduces the average identifier size needed for logging.

The data presented in Figure 3 only takes into account the average token size. Since token size varies for bit aligned identifiers with scoping and different tokens appear with different frequencies at runtime, the observed average size of tokens in the runtime log may not match that derived above. We continue our analysis by gathering runtime logs from 17 logging cases: each of 13 TinyOS subsystem in isolation and 4 large combinations of subsystems. For each gathered log we calculate the observed average token width appearing in the runtime logs for bit aligned identifiers with scoping. Table II lists this observed average token size, along with the byte aligned and bit aligned sizes that can be calculated precisely. Observed performance of logs gathered using LIS with token scoping outperforms that predicted by Figure 3. This is because the static token identifier size calculation ignores the high frequency at which some short tokens occur.

This evaluation shows that the combination of bit aligned logging and name space scoping provided by LIS creates significantly more compact runtime logs than either byte aligned or bit aligned logging without token scoping.

### C. Static and Runtime Overheads from Bit Aligned Logging

Using bit aligned identifiers from multiple scoped name spaces provides attractive reductions in log bandwidth, but use of such a technique must respect static and runtime constraints of the system on which it is used. We evaluated bit aligned logging by measuring the overhead of our `bitlog` library.

The `bitlog` library is implemented using standard ANSI C features and is easily portable to new platforms that have a C compiler. Logging overhead comes from the static costs of introducing logging statements and the support library into the instrumented system, and from the runtime cost associated with calling into the logging infrastructure.

Table III lists the program memory and RAM overheads of the `bitlog` library, the `LogTap` TinyOS component, and the surrounding TinyOS communication and routing stack for the MicaZ sensor node using the AVR ATMeag128 processor. Also listed in the table is the overhead resulting from adding

TABLE IV
CYCLE COUNTS OF LIS LOGGING ACTIONS

| Action | Avg. | Min. | Max. |
|---|---|---|---|
| `bitlog_init` | 99.0 ± 0.0 | 99 | 99 |
| `bitlog_write` | 176.5 ± 1.5 | 176 | 241 |
| `bitlog_flush` | 1572.5 ± 612.5 | 1319 | 3130 |

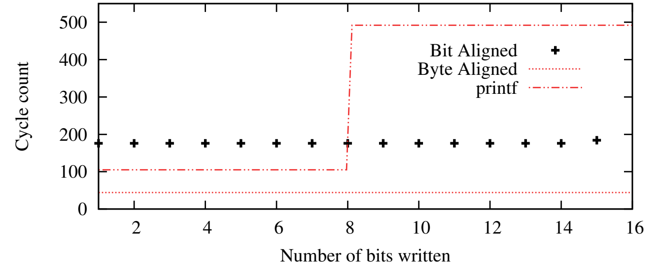


Fig. 4. Range of cycle counts observed from calling various logging functions. Bit aligned logging using `bitlog_write` requires about 1.7 times as long as `printf` (for one uninterpreted byte) and 4 times that of direct array access. This increase in latency is a cost of lower bandwidth logs.

a single call to the `bitlog_write` function provided by `bitlog` and used by LIS to log tokens.

Latency is introduced into the systems as a result of initializing the logging system with a call to `bitlog_init`, logging data by calling `bitlog_write`, and flushing the log to the `LogTap` component through calls to `bitlog_flush`. We profiled the latency introduced by these calls using the cycle accurate Avrora simulator [6]. The `bitlog_flush` function is automatically called in `bitlog_write` when the log buffer fills. To isolate the overhead of each function, we used preemptive flushes of the log buffer to prevent `bitlog_write` from triggering a flush during our evaluation.

Table IV lists the cycle counts required to execute each function within the `bitlog` library running on a lightly loaded system. The minimum number of cycles observed is the result of calling the function when no interrupts occur and a flush is not triggered. Deviations from this minimum are the result of interrupts (primarily from the timer unit) firing during function execution. Of particular interest is the expected and maximal costs of calling `bitlog_write`, since this is the logging call used by LIS to perform binary aligned logging.

Figure 4 compares the average cycles required for calls to `bitlog_write` to those observed for alternate logging strategies. We evaluated byte aligned logging by writing 8-bit or 16-bit values directly into a byte array. The `printf` function was evaluated by writing one or two bytes of constant string data using the TinyOS `printf` implementation. Bit aligned logging requires an average of 177 cycles, about 4 times the overhead of byte aligned logging and a 1.7 times that used by `printf` to write a byte of constant data. We feel this modest latency increase is acceptable for most embedded systems, which are bandwidth bound rather than CPU bound. An exception to this are real time systems where additional overhead from any form of logging may be unacceptable.

Worst case latency introduced into an instrumented system arises when `bitlog_write` triggers a call to `bitlog_flush`. The additional overhead from `bitlog_flush` is primarily caused from copying the log buffer out of the `bitlog` library. This overhead will be present in any logging system that maintains isolated buffers that are copied into the underlying operating system. This latency can be mitigated by preemptively flushing log buffers at non-critical points of the program. Alternatively, the `bitlog` library can be integrated into the underlying log management system to eliminate copying. But this integration comes at the cost of decreased portability.

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.

### D. Discussion

Our evaluation shows that the benefits of bit aligned logging are most significant when the logged identifiers are quite small. The use of token scoping provided by LIS maintains small identifier sizes, even as the number of tokens handled by the entire system increases. This results in significant reductions to the bandwidth needed by a given logging task. Bit aligned logging can be implemented in a manner that honors the resource constraints of the target platform. The current implementation of LIS has been used without problems on 8-bit and 16-bit microcontrollers running TinyOS. While logging in time critical systems requires an awareness of logging overheads, we have found LIS to be a valuable asset to understanding problems both in and out of timing sensitive systems.

## 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. 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 [7] and Event Tracing for Windows [8]. 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 data and extract meaningful features [9]–[11]. Unfortunately, the underlying premise to “log everything all the time” overwhelms the limited resources available on embedded systems and restricts the direct applicability of desktop or cluster systems solutions within 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). ICE and ICD provide detailed insight to an individual device, but typically require costly external hardware and often require physical access to the embedded system. These requirements make ICE and ICD most effective for pre-deployment testing of individual hardware components.

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 [12], precisely model the underlying hardware [6], or model physical phenomenon such as the radio channel [13]. Each of these points caters towards solving a specific class of problem and is important for initial system development. Unfortunately, problems not seen during simulation often appear during deployment.

Wireless embedded systems present additional design challenges due to their limited communication bandwidth (often less than 1 Mbps), extensive interaction between devices, and the close coupling between individual devices and the physical world.

One approach targeted for wireless systems is to collect logs using a secondary network of “sniffer” nodes [14]–[16]. Such an approach would work well along side LIS and provide additional depth to the logs LIS collects.

LIS specializes in logging runtime state, a feature that other tools can reuse. Sympathy [17] 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 [18] combines logging with an extension of the Apriori [19] data mining algorithm 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.

The sensor network management system [1] also logs runtime state, but does so through a free form `printf` interface. In contrast to this type of logging framework, LIS provides a more structured logging framework that separates logging specifications from application code and that facilitates aggressive log bandwidth optimization.

Some tools [20]–[22] actively reason about runtime state while watching for correctness violations. Other frameworks allow the value of data to not only be observed, but also to be manipulated [23]–[25]. Declarative frameworks [26]–[28] provide an intriguing mix between the active fault monitoring tools and the more passive logging infrastructures. 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.

Despite ongoing research, most of the tools discussed within this section have yet to find widespread use within the wireless sensor network community. More commonly observed is the humble and ubiquitous “LED debugging”. LED debugging provides developers with a log maintained in the head of an observer carefully counting LED blinks or frantically scratched

onto a scrap of paper. The success of LED debugging is most probably due to its ease of use. 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 ease of use with efficient bandwidth usage to make it a great alternative to littering code with blinking LEDs and verbose calls to `printf`.

## V. Conclusions

This research explores the significant reductions to log bandwidth achievable through the use of bit aligned identifiers formed from scoped token sets. We highlight the need for extremely small identifiers if significant gains are to be had from bit aligned identifier logging. Scoping of logged tokens into isolated name spaces facilitates the creation of extremely small token identifiers. LIS provides a clean interface for developers to use to integrate these ideas into their daily work. We hope that this work motivates further research into logging infrastructures that provide developers alternatives to ad-hoc and error prone “LED debugging” approaches to 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 these limitations can be mediated by future expansions to the LIS infrastructure using established compiler and program analysis techniques.

LIS focuses on providing exceptional 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 does not natively support revising program state or extending program functionality. 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.

We’ve found LIS to provide quick and effective logging support for a wide range of diagnostic tasks we encounter from our daily work with wireless embedded devices. The complete LIS framework and infrastructure for integration into the TinyOS build system is available online at https://projects.nesl.ucla.edu/~rshea/lis/.

## Acknowledgments

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 reflect the views of the listed funding agencies.

## References

[1] G. Tolle and D. Culler, “Design of an application-cooperative management system for wireless sensor networks,” in *EWSN*. IEEE, 2005.
[2] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, “The nesc language: A holistic approach to networked embedded systems,” in *PLDI*, 2003.
[3] R. Fonseca, O. Gnawali, K. Jamieson, P. L. Sukun Kim, and A. Woo, “The collection tree protocol,” TEP, Tech. Rep. 123, 2006.
[4] G. Necula, S. McPeak, S. P. Rahul, and W. Weimer, “Cil: Intermediate language and tools for analysis and transformation of c programs.”
[5] R. Shea, M. Srivastava, and Y. Cho, “Optimizing bandwidth of call traces for wireless embedded systems,” in *Embedded Systems Letters*. IEEE, 2009.
[6] B. Titzer, D. Lee, and J. Palsberg, “Avrora: Scalable sensor network simulation with precise timing,” in *IPSN*. ACM, 2005.
[7] K. Yaghmour and M. Dagenais, “Measuring and characterizing system behavior using kernel-level event logging,” in *USENIX*. USENIX, 2000.
[8] *Event Tracing for Windows*, Microsoft Corporation, 2008.
[9] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, “Magpie: Online modelling and performance-aware systems,” in *HotOS*. USENIX, 2003.
[10] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, “Capturing, indexing, clustering, and retrieving system history,” *SIGOPS*, 2005.
[11] 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.
[12] P. Levis, N. Lee, M. Welsh, and D. Culler, “Tossim: Accurate and scalable simulation of entire tinyos applications,” in *SenSys*. ACM, 2003.
[13] *QualNet User Manual v 4.5*, 2008.
[14] B.-R. Chen, G. Peterson, G. Mainland, and M. Welsh, “Livenet: Using passive monitoring to reconstruct sensor network dynamics,” in *DCOSS*. Springer-Verlag, 2008.
[15] K. Römer, “Passive distributed assertions for sensor networks,” in *DCOSS*. Springer-Verlag, 2008.
[16] M. M. H. Khan, L. Luo, C. Huang, and T. Abdelzaher, “Snts: Sensor network troubleshooting suite,” in *LNCS*. Springer, 2007.
[17] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin, “Sympathy for the sensor network debugger,” in *SenSys*. ACM, 2005.
[18] M. M. H. Khan, H. K. Le, H. Ahmadi, T. F. Abdelzaher, and J. Han, “Dustminer: Troubleshooting interactive complexity bugs in sensor networks,” in *SenSys*. ACM, 2008.
[19] R. Agrawal and R. Srikant, “Fast algorithms for mining association rules,” in *VLDB*, 1994.
[20] V. Krunic, E. Trumpler, and R. Han, “Nodemd: diagnosing node-level faults in remote wireless sensor systems,” in *MobiSys*. ACM, 2007.
[21] W. Archer, P. Levis, and J. Regehr, “Interface contracts for tinyos,” in *IPSN*. New York, NY, USA: ACM, 2007, pp. 158–165.
[22] R. Kumar, E. Kohler, and M. Srivastava, “Harbor: software-based memory protection for sensor nodes,” in *IPSN*. ACM, 2007.
[23] 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.
[24] L. Luo, T. He, , G. Zhou, L. Gu, T. Abdelzaher, and J. Stankovic.
[25] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse, “Clairvoyant: a comprehensive source-level debugger for wireless sensor networks,” in *SenSys*. ACM, 2007.
[26] 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.
[27] A. Tavakoli, “Wringer: A debugging and monitoring framework for wireless sensor networks,” in *SenSys Doctoral Colloquium*. ACM, 2007.
[28] 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.