Log Instrumentation Specifications and Low Overhead Logging

Roy Shea

Department of Computer Science
University of California, Los Angeles
roy@cs.ucla.edu

July 17, 2009

With Mani Srivastava, Young Cho, and endless input from folks in NESL.
Observing the Unobservable

The Vision

In-network algorithms directing a sensor network made up of thousands (or more) of sub-dollar wireless sensor nodes to localize a lion roaming through the jungle.
Observing the Unobservable

The Vision

In-network algorithms directing a sensor network made up of thousands (or more) of sub-dollar wireless sensor nodes to localize a lion roaming through the jungle.

Problems with the Vision
Observing the Unobservable

The Vision

In-network algorithms directing a sensor network made up of thousands (or more) of sub-dollar wireless sensor nodes to localize a lion roaming through the jungle.

Problems with the Vision

- Lions live in the savanna
Observing the Unobservable

The Vision

In-network algorithms directing a sensor network made up of thousands (or more) of sub-dollar wireless sensor nodes to localize a lion roaming through the jungle.

Problems with the Vision

- Lions live in the savanna
- Still paying $139.00 for a sensor node (TelosB with sensor suite)
Observing the Unobservable

The Vision

In-network algorithms directing a sensor network made up of thousands (or more) of sub-dollar wireless sensor nodes to localize a lion roaming through the jungle.

Problems with the Vision

- Lions live in the savanna
- Still paying $139.00 for a sensor node (TelosB with sensor suite)
- “Thousands” is a big number
Observing the Unobservable

The Vision

In-network algorithms directing a sensor network made up of thousands (or more) of sub-dollar wireless sensor nodes to localize a lion roaming through the jungle.

Problems with the Vision

- Lions live in the savanna
- Still paying $139.00 for a sensor node (TelosB with sensor suite)
- “Thousands” is a big number
- Easy to create suboptimal, fragile, and buggy localized algorithms
After the Dream is Gone

Picking Up Pieces

- Lions live in the savanna
  - Embrace breadth of expertise while seeking out researchers in other fields
Picking Up Pieces

- Lions live in the savanna
  - Embrace breadth of expertise while seeking out researchers in other fields
- Still paying $139.00 for a sensor node (TelosB with sensor suite)
  - Count on the power of economy of scale *if* the demand is present
After the Dream is Gone

Picking Up Pieces

- Lions live in the savanna
  - Embrace breadth of expertise while seeking out researchers in other fields
- Still paying $139.00 for a sensor node (TelosB with sensor suite)
  - Count on the power of economy of scale if the demand is present
- “Thousands” is a big number
  - Ends up that you can learn a lot about the world with one node, or two nodes, or ten nodes...
After the Dream is Gone

Picking Up Pieces

- Lions live in the savanna
  - Embrace breadth of expertise while seeking out researchers in other fields
- Still paying $139.00 for a sensor node (TelosB with sensor suite)
  - Count on the power of economy of scale if the demand is present
- “Thousands” is a big number
  - Ends up that you can learn a lot about the world with one node, or two nodes, or ten nodes...
- Easy to create suboptimal, fragile, and buggy localized algorithms
  - Provide tools to help developers write efficient, robust, and functioning code for wireless embedded systems
Observing the Unobservable
Within the Observers of the Unobservable

My Vision

Develop new techniques and tools that provide developers with the insight needed to create applications for and understand behavior within deployed networks of bottom tier sensing devices.
Observing the Unobservable
Within the Observers of the Unobservable

My Vision

Develop new techniques and tools that provide developers with the insight needed to create applications for and understand behavior within deployed networks of bottom tier sensing devices.

LIS
Framework for describing and implementing logging tasks.

LowLog
Optimizing to token name spaces to reduce the overhead of collecting runtime call traces.
Machine Class

Desktop / Server

Embedded Computer

Microcontroller

High End
- Native hardware support for debugging
- Ample resources that debugging infrastructure can use
- Rich I/O capabilities

Bottom Tier
- Real time constraints limit interactive or CPU intensive debugging techniques
- Minimal resources require very small footprint debugging utilities

Roy Shea (UCLA)
Number of Machines (or Processes)

- Basic debugging setup
- Ability to suspend program without impacting other processes

Suspension of multiple processes tricky
- Interactive debugging tracks execution on multiple processes to understand interactions
- Logs capturing consistent system state are a powerful debugging aid

- Suspending execution becomes very difficult
- Challenging to capture consistent system state
- Use detailed logs used to understand aggregate system behavior
Debugging Complexity of Different Systems

- Microcontroller
- Embedded Computer
- Desktop / Server

- One Device
- Two Devices
- Fewer than 10 Devices
- 10 or more Devices
Handling Bugs Through the Entire System Design Cycle

- Interactie Debugging
- Testing
- Monitoring
- Simulation
- Logging

- Specification and Verification

- Design
- Controlled Execution
- Deployed Execution
Outline

1. Introduction
2. Log Instrumentation Specifications
3. Low Overhead Logging with LIS
4. Conclusions
Observe demand for `printf` style logging in distributed embedded systems

- Low learning curve and direct semantics make it easy to use
- But resulting logs are typically verbose and ad-hoc logging is difficult to maintain

Want to provide developers an alternate logging solution

- Separation of logging specification from underlying code base
- Encourages design of optimized logging tasks
- Easy to pickup and use

Logging streams of tokens via LIS is our answer
Architecture

Original Application

OS and Application

Program Generated LIS Script

or

User Generated LIS Script

LIS Instrumentation Engine

Parser

Token Selection

Instrumentation

LIS Instrumented Application

OS and Application

libbitlog

Log Manager

User Application
Architecture

Original Application

OS and Application

Program Generated LIS Script

User Generated LIS Script

or

LIS Instrumentation Engine

Parser

Token Selection

Instrumentation

LIS Instrumented Application

OS and Application

libbitlog

Log Manager

LIS Core

LIS Script

Log Manager

User Application
Structure of a LIS Statement

**Statement Types**
- Function header
- Function footer
- Function call
- Control flow
- Variable watchpoint

**Statement Location**
- Union of log type and a function name
- Type specific directives bind log to specific called function, control flow type, or variable name
Structure of a LIS Statement

**Statement Types**
- Function header
- Function footer
- Function call
- Control flow
- Variable watchpoint

**Statement Location**
- Union of log type and a function name
- Type specific directives bind log to specific called function, control flow type, or variable name

**Token Scope**

- **Global**  Token value unique throughout the program
- **Local**   Token value is unique throughout the function
- **Point**   Token value is a singled fixed value
LIS Specification

\[
\begin{align*}
\text{Start} & \rightarrow \text{Statements} \mid \epsilon \\
\text{Statements} & \rightarrow \text{Stmt} \ \text{Statements} \mid \text{Stmt} \\
\text{Stmt} & \rightarrow \text{Header} \mid \text{Footer} \mid \text{Call} \mid \text{ControlFlow} \mid \text{Watch} \\
\text{Header} & \rightarrow \text{header} \ \text{Placement} \ \text{Scope} \\
\text{Footer} & \rightarrow \text{footer} \ \text{Placement} \ \text{Scope} \\
\text{Call} & \rightarrow \text{call} \ \text{Placement} \ \text{Scope} \ \text{Target} \\
\text{ControlFlow} & \rightarrow \text{controlflow} \ \text{Placement} \ \text{Scope} \ \text{Flag} \ \text{Var} \\
\text{Watch} & \rightarrow \text{watch} \ \text{Placement} \ \text{Scope} \ \text{Var} \\
\text{Placement} & \rightarrow F \\
\text{Scope} & \rightarrow \text{global} \mid \text{local} \mid \text{point} \\
\text{Target} & \rightarrow F \mid _\text{PTR}_ \\
\text{Flag} & \rightarrow \text{if} \mid \text{switch} \mid \text{loop} \mid \text{if} - \text{switch} \mid \text{if} - \text{loop} \\
& \quad \mid \text{switch} - \text{loop} \mid \text{if} - \text{switch} - \text{loop} \\
\text{Var} & \rightarrow \langle \text{Variable name from program} \rangle \mid _\text{ANY}_ \\
F & \rightarrow \langle \text{Function name from program} \rangle
\end{align*}
\]
```c
/* 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;
}
Instantiation of LIS

Instrumentation Engine

- Built using C Intermediary Language (CIL) framework
  - Input a LIS script and source program
  - Outputs instrumented program
- Functional ports for x86, ATMega128, and MSP430 chip sets
- Patch available to integrate LIS into the TinyOS build system
Instantiation of LIS

Instrumentation Engine

- Built using the C Intermediary Language (CIL) framework
  - Input a LIS script and source program
  - Outputs instrumented program
- Functional ports for x86, ATMega128, and MSP430 chip sets
- Patch available to integrate LIS into the TinyOS build system

<table>
<thead>
<tr>
<th></th>
<th>Mica2</th>
<th>MicaZ</th>
<th>TelosB</th>
</tr>
</thead>
<tbody>
<tr>
<td>TinyOS Radio Stack</td>
<td>7178</td>
<td>9264</td>
<td>8456</td>
</tr>
<tr>
<td>TinyOS CTP</td>
<td>9692</td>
<td>10284</td>
<td>11126</td>
</tr>
<tr>
<td>LogTap (CTP)</td>
<td>1384</td>
<td>1412</td>
<td>2228</td>
</tr>
<tr>
<td>LogTap (broadcast)</td>
<td>108</td>
<td>74</td>
<td>388</td>
</tr>
<tr>
<td>Bitlog Library</td>
<td>386</td>
<td>386</td>
<td>362</td>
</tr>
<tr>
<td>Call to bitlog_write</td>
<td>14</td>
<td>14</td>
<td>12</td>
</tr>
</tbody>
</table>
Architecture

Original Application

OS and Application

Program Generated LIS Script

User Generated LIS Script

LIS Instrumentation Engine

Parser

Token Selection

Instrumentation

LIS Instrumented Application

LIS Core

LIS Script

Log Manager

User Application

libbitlog

Log Manager
## Instantiation of LIS

### Runtime Support

### Bitlog Library
- Writing of tokens into a buffer managed by a logging library
- Includes the Bitlog library that provides low overhead bit aligned logging

### LogTap
- Storage / transfer of logs managed by log management library
- Writing log buffers out to stderr for use on desktop machines
- Broadcasting logs using the TinyOS AMSend interface
- Routing logs using the TinyOS collection tree protocol (CTP)
Instantiation of LIS

Runtime Support

![Graph showing cycle count vs number of bits written. The graph compares different conditions: Flush, No interrupts (blue line), No flush, No interrupts (red line). The y-axis represents cycle count, and the x-axis represents the number of bits written.](image-url)
Using LIS: Observing CRC Errors

controlflow CC2420ReceiveP.RXFIFO.readDone global if buf
Using LIS: Observing CRC Errors

Distance between nodes (meters)

Time (seconds)

CRC Error on Base Station
CRC Error on Mobile Node
Using LIS: Learning about CTP

header CtpForwardingEngineP.0.sendTask.runTask point controlflow CtpForwardingEngineP.0.sendTask.runTask \ local if–switch–loop __ANY__
Using LIS: Learning about CTP

```
header CtpForwardingEngineP.0.sendTask.runTask point
controlflow CtpForwardingEngineP.0.sendTask.runTask \\
local if-switch-loop __ANY__
```

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

Roy Shea (UCLA)
Outline

1. Introduction
2. Log Instrumentation Specifications
3. Low Overhead Logging with LIS
4. Conclusions
Tracing Calls Within a Region of Interest with LowLog

- Function call traces
  - Describe the sequence of function calls made at runtime
  - Already used by developers to understand system behavior
  - Often provides enough information in itself to diagnose bugs or jump start more aggressive debugging efforts

- Region of interest (ROI)
  - Subsystem of interest to a developer
  - For example there is (usually) no reason to trace execution through the kernel when debugging a user space application
  - Provides a focused view of a system

- LowLog provides optimized region of interest call tracing
- LowLog sits as a higher level analysis that outputs efficient LIS scripts
Partitioning a Program into an ROI

- *Entry function* is reachable from a function not within the ROI
- *Body function* only reachable from functions within the ROI
- *Return of interest* when any function from the ROI returns
Partitioning a Program into an ROI

- *Entry function* is reachable from a function not within the ROI
- *Body function* only reachable from functions within the ROI
- *Return of interest* when any function from the ROI returns
Partitioning a Program into an ROI

- *Entry function* is reachable from a function not within the ROI
- *Body function* only reachable from functions within the ROI
- *Return of interest* when any function from the ROI returns
Create Log of Entry, Body, and Return Tokens
Standard call tracing uses *global identifier logging* that lumps entry and body tokens into a single name space and use simple fixed bit width token encoding.

LowLog proposes to alternate schemes:

- **Local identifier logging** creates multiple caller specific token name spaces
- **Control flow logging** tracks runtime control flow decisions rather than body calls

All three logging techniques can apply more powerful encoding techniques if more information is available.
Only Log Control Flow Decisions Effecting Called Functions

- Reverse dataflow analysis to track sets of functions that must be called after reaching a statement
- Join function examines sets of called functions
  - If the sets are different then logging of the control flow taken form the current node is tracked and the empty set is passed into the transfer function
  - If the sets are identical then the set (either since they are identical) is passed into the transfer function
- Transfer function prepends called functions from the current node to the list of called functions
Call Trace Bandwidth

![Bar chart showing bandwidth required for different identifiers and flows.]

- Global Identifier: 600 B/s
- Huffman (Global): 200 B/s
- Local Identifier: 100 B/s
- Control Flow: 50 B/s
- Huffman (Local): 25 B/s

Roy Shea (UCLA)
Effect of ROI Size on Token Width

Average token size (bits) vs. Number of functions in ROI

- Global ID Header Token
- Local ID Header Token
- Local ID Call Token

Roy Shea (UCLA)
LIS
July 17, 2009 31 / 36
Effect of ROI Size on Bandwidth

![Graph showing the effect of ROI size on bandwidth. The graph compares Global ID Logging and Local ID Logging. The x-axis represents the number of functions in ROI, and the y-axis represents log bandwidth (bits/sec). The graph shows a significant increase in bandwidth as the number of functions in ROI increases.](image-url)
Using LowLog: Problems in the Radio Stack

CC2420 Transmit P
Using LowLog: Problems in the Radio Stack

CC2420TransmitP

- TinyOS runtime resource accounting and message time stamping each work in isolation
- Combining the two causes many time stamps to be lost
- Gathered call traces from the TinyOS CC2420TransmitP component
  - Traces with both resource accounting and time stamping reveal a frequent number of calls to CC2420Receive.sfd
  - Insight limited code search down to about 20 lines of code
- Timing delays introduced by resource accounting cause an overly conservative block of code to invalidate time stamps
Outline

1 Introduction

2 Log Instrumentation Specifications

3 Low Overhead Logging with LIS

4 Conclusions
Summary

- LIS and LowLog help developers understand what their system is doing
  - LIS provides developers with a convenient and powerful framework for describing logging tasks
  - LowLog sits upon LIS and provides optimized call trace logging
- Have not yet found any lions, but am still looking

Want to play with LIS and LowLog?

http://nesl.ee.ucla.edu/research/lis
Questions?