Scoped Identifiers for Efficient Bit Aligned Logging

Roy Shea¹  Mani Srivastava²  Young Cho³

¹Department of Computer Science
University of California, Los Angeles

²Departments of Electrical Engineering and Computer Science
University of California, Los Angeles

³Information Sciences Institute
University of Southern California

March 11, 2010
Overview

- Participated in developing research oriented distributed embedded systems
- Software defects proved very hard to diagnose
  - Standard diagnostic techniques were poor fit
  - Lacked tools to view runtime state of distributed embedded system
- Motivated designing a new logging framework that respects the bandwidth limitations of the distributed embedded domain
Debugging Distributed Systems is Hard

- Algorithm Design
- Device Independent Implementation
- Device Dependent Implementation
- Reduced Scale Testing
- Full Scale Deployment
Debugging Distributed Systems is Hard

Fidelity

- Algorithm Design
- Device Independent Implementation
- Reduced Scale Testing
- Full Scale Deployment

Visibility

- Device Dependent Implementation
Why Logging?

Runtime Solution
Because defects emerged during deployment despite good design practices

Passive Solution
Because interactive techniques have limited applicability to domain

Runtime logs explain unexpected system behavior without user interaction during deployment
Growing Beyond printf

- Logging has a long history of printf style interfaces
  - Immediately understandable to developer
  - Easy to integrate into regular work flow

Growing Beyond printf
Growing Beyond printf

- Logging has a long history of printf style interfaces
  - Immediately understandable to developer
  - Easy to integrate into regular work flow
-Verbose free form logging stresses bandwidth in distributed embedded systems
-External research re-encodes ASCII strings with numeric identifiers to reduced log bandwidth
Savings From Bit Aligned Logging
Savings From Bit Aligned Logging
Savings From Bit Aligned Logging
Identifiers are typically assigned from a global name space.

21 tokens in a single name space requires 5-bits to encode.
Real Programs Have Rich Structure

Exploit program structure to more compactly encode tokens
Local Token Scoping for Short Identifiers

3 tokens $\rightarrow$ 2 bits

6 tokens $\rightarrow$ 3 bits

1 token $\rightarrow$ 1 bit
Log Instrumentation Specifications (LIS)

- Scripting language to describe logging tasks
- Generalized logging solution providing explicit scoping declarations

  **Global** tokens assigned identifiers unique across all tokens
  **Local** tokens assigned identifiers unique within a function
  **Point** token uses a single well defined identifier
LIS Scripts

- LIS scripts reside outside of the code base
- Script is composed of statements describing runtime state to be gathered
  - Base statement types target key function properties
  - Combination of statement type, function name, and optional modifiers control location of instrumentation
- LIS scripts are written by developers or emitted by higher level analysis
Demonstrative Logging Task

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


**Demonstrative Logging Task**

```c
void read_done(error_t result, uint16_t data) {
    bitlog_write(4, 3); /* Log header */
    if (send_busy == TRUE) {
        return;
    }
    /* Rest of function body elided... */
    return;
}
```
**Demonstrative Logging Task**

```c
void read_done(error_t result, uint16_t data) {
    bitlog_write(4, 3); /* Log header */
    if (send_busy == TRUE) {
        bitlog_write(5, 3); /* Log control flow */
        return;
    }
    bitlog_write(6, 3); /* Log control flow */

    /* Rest of function body elided... */
    return;
}
```
Demonstrative Logging Task

```c
void read_done(error_t result, uint16_t data) {
    bitlog_write(4, 3); /* Log header */
    if (send_busy == TRUE) {
        bitlog_write(5, 3); /* Log control flow */
        bitlog_write(0, 1); /* Log footer */
        return;
    }
    bitlog_write(6, 3); /* Log control flow */
    /* Rest of function body elided... */
    bitlog_write(0, 1); /* Log footer */
    return;
}
```
Demonstrative Logging Task

```c
void read_done(error_t result, uint16_t data) {
  bitlog_write(4, 3);  /* Log header */
  if (send_busy == TRUE) {
    bitlog_write(5, 3);  /* Log control flow */
    bitlog_write(0, 1);  /* Log footer */
    return;
  }
  bitlog_write(6, 3);  /* Log control flow */
  /* Rest of function body elided... */
  bitlog_write(0, 1);  /* Log footer */
  return;
}
```
Evaluation Background

- Implemented small stand alone bitlog library
- LIS framework developed to instrument C source code
- Evaluation targets the ATMega128 microcontroller and CC2420 radio
- Cycle counts obtained using the cycle accurate Avrora simulator
- Bandwidth measurements obtained using small testbed
# Static Costs of Bit Aligned Logging

<table>
<thead>
<tr>
<th>System Component</th>
<th>Program Memory (bytes)</th>
<th>Data Memory (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Radio Stack</td>
<td>9264</td>
<td>210</td>
</tr>
<tr>
<td>Routing Layer</td>
<td>10284</td>
<td>1360</td>
</tr>
<tr>
<td>Routed Log Management</td>
<td>1412</td>
<td>351</td>
</tr>
<tr>
<td>Broadcast Log Management</td>
<td>74</td>
<td>128</td>
</tr>
<tr>
<td>Bitlog Library</td>
<td>290</td>
<td>24</td>
</tr>
<tr>
<td>Call to bitlog_write</td>
<td>14</td>
<td>0</td>
</tr>
</tbody>
</table>
Latency from Bit Aligned Logging

![Graph showing latency from bit aligned logging. The x-axis represents the number of bits written, and the y-axis represents the cycle count. There are three lines: Bit Aligned, Byte Aligned, and printf. The Bit Aligned line shows a sharp increase in cycle count at a certain number of bits written, while the Byte Aligned and printf lines remain relatively flat.](image-url)
Number of Tokens Affects Bandwidth

![Graph showing the relationship between the number of functions being traced and log bandwidth for global name space and local token scoping.](image-url)
## Observed Run Time Identifier Size

<table>
<thead>
<tr>
<th>Alignment</th>
<th>Scoping</th>
<th>Average Size (bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>&lt; 60 Tokens</td>
</tr>
<tr>
<td>Byte</td>
<td>—</td>
<td>8.0 ± 0.0</td>
</tr>
<tr>
<td>Bit</td>
<td>Global</td>
<td>3.5 ± 1.6</td>
</tr>
<tr>
<td>Bit</td>
<td>Local</td>
<td>2.8 ± 0.5</td>
</tr>
</tbody>
</table>
Number of Tokens Affects Identifier Width

![Graph showing the relationship between the number of functions being traced and the average token width. The graph includes different token scoping types: "Entry" Tokens (yellow triangles), "Body" Tokens (red crosses), and Global Name Space (green plus signs). The y-axis represents the average token width in bits, while the x-axis represents the number of functions being traced.](image)

Local Token Scoping "Entry" Tokens
Local Token Scoping "Body" Tokens
Global Name Space
Limitations

- Current LIS implementation bases scopes on function boundaries
- Token scoping does not compress program data
- Care must be taken to guarantee parsable logs
  - Parsing depends on knowing the local context
  - Script writing guidelines guarantee parsable logs
Conclusions

- Established technique of bit aligning data used to reduce log bandwidth...
  - but only effective with consistently small identifiers
- Local *token scoping* can produce consistently small token identifier encodings...
  - but requires a great deal of error prone book keeping
- LIS automates the difficulties of token scoping and provides a general interface for describing logging tasks
Questions?

https://projects.nesl.ucla.edu/~rshea/lis/