Optimizing Bandwidth of Call Traces for Wireless Embedded Systems

Roy Shea, Mani Srivastava *Senior Member, IEEE,* and Young Cho *Member, IEEE*

***Abstract*—Call traces expose runtime behaviors that greatly aid system developers in profiling performance and diagnosing problems within wireless embedded applications. Strict resource constraints limit the volume of trace data that can be handled on embedded devices, especially bandwidth limited wireless embedded systems. We propose two new call trace gathering techniques, local identifier logging and control flow logging, which provide significant reductions in bandwidth consumption compared to the current standard practice of global identifier logging. Intuition into the savings made possible by the proposed trace gathering techniques is provided by an analytical comparison of the bandwidth required by various call tracing approaches. Confirmation of this intuition is demonstrated through experimentation that reveals log bandwidth savings of approximately 85% compared to global identifier logging using flat name spaces, and 35% compared to global identifier logging using optimal Huffman coding.**

***Index Terms*—Wireless embedded systems, logging, bandwidth compression, dataflow.**

I. Introduction

Traditional embedded systems debugging tools, such as JTAG, fail in the wireless embedded domain where direct hardware access is often not available and where complex component interactions limit the utility of single device debugging. Further stumbling blocks are added by the real time nature of wireless embedded environments, where an ever changing environment and dynamic interactions with neighbors limit the applicability of abstractions such as break points.

Logging has long been used to monitor state and diagnose problems in traditional systems. New logging techniques [1], [2] provide ever greater insight into single systems, while advances in distributed monitoring [3], [4] continue to increase our understanding and ability to debug distributed protocols. These works provided guidance in what to log and how logs can be used, but are designed for environments without the bandwidth constraints of wireless embedded systems. While simulated [5], [6] and testbed [7], [8] wireless embedded deployments grant significant insight into a system and provide bandwidth rich back channels for logging, experience repeatedly teaches us that migration of a wireless system into the field almost always requires diagnosing new sets of problems in-situ. Recent sensor network research explores the role of logging in resource constrained settings by examining how best to collect and expose logs to understand runtime behaviors and diagnose runtime problems [9]–[14].

Our research addresses the bandwidth bottleneck that limits adoption of logging in wireless embedded systems by optimizing the bandwidth efficiency of gathering function call trace logs. Function call traces track the order and nesting of calls made by a device at runtime, and provide insight critical for making informed decisions about system performance and diagnosing unexpected behavior. These logs are of great value to both developers and analyses [15]–[17].

Developers often wish to know the behavior of a specific region of code or suspect that a bug originates from a particular subsystem, and call traces can be easily limited to such a *region of interest* (ROI). An ROI is a set of functions selected by a developer as being of interest and though which execution is to be logged at runtime, while functions outside the ROI execute without modifying the log. Function call traces are commonly constructed from logs of unique function identifiers, a technique we refer to as *global identifier logging*. Such a log is formed by extending each function within an ROI to log a unique identifier when entered and to log a short return identifier when the function returns.

Identifier assignment is critical for creating an efficient logging framework and is the focus of this research. Global identifier logging uses a globally unique identifier for each function within the ROI. Presented in this paper are two alternate strategies, *local identifier logging* and *control flow logging*, that use multiple small identifier name spaces to allow shorter identifiers and thus more compact call trace logs.

In recent work, we explore a new specification language to describe what to log within an embedded system and briefly examined using this language to gather call traces using multiple name spaces [18]. This research presents an analytical model to compare call trace gathering techniques, introduces control flow logging facilitated by a dataflow analysis to optimize control flow log points, and evaluates the logging mechanisms using both fixed bit width and Huffman coding to encode identifiers.

II. Intuition Behind Optimizing Call Traces

Our proposed alternate logging schemes reduce call trace size relative to global identifier logging by reducing the number of bits required to encode identifiers. The reductions are best understood by sorting identifiers, or tokens, of a call trace into three classes:

- *Entry tokens* for *entry calls* record calls from outside to within the ROI.
- *Body tokens* for *body calls* record calls between functions within the ROI.
- *Return tokens* record returning from a function within the ROI.

Local identifier logging creates smaller identifier name spaces by separating the entry tokens from the body tokens and by giving each function within the ROI its own name space from which to assign body call token identifiers. Control flow logging uses control flow information to further partition each function's local body token name space, providing benefits over local identifier logging when chains of function calls with minimal interspersed control flow are encountered.

III. Formalization

We model the bandwidth required by the three logging techniques using a formal abstraction of a program, regions of interest, and

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.

Roy Shea is with the Computer Science department, University of California, Los Angeles, CA 90095-1594 (email: roy@cs.ucla.edu).

Mani Srivastava is with the Electrical Engineering and Computer Science departments, University of California, Los Angeles, CA 90095-1594 (email: mbs@ucla.edu).

Young Cho is with the Information Sciences Institute, University of Southern California, Los Angeles, CA 90292-6611 (email: youngcho@isi.edu).

TABLE I
FORMALIZATION OF A PROGRAM, REGIONS OF INTEREST, AND RUNTIME PROGRAM BEHAVIOR.

	Notation	Description
Program	F	Set of functions in program P.
	G_f	Graph representation of function f described by the set N_f of basic blocks and the edge matrix E_f denoting links between basic blocks.
Region of Interest	R	Set of functions in a region of interest. R is partitioned by the set of entry function R_e and the set of body functions R_b.
	C_{f_R}	Set of functions within R that are called by function f.
Runtime Behavior	B	Call frequency matrix where entry b_{ij} describes the average frequency at which function i calls function j within the program.
	CFG	Set of matrices tracking the frequency at which the transition from a basic block i to the basic block j within a function f is taken.

runtime program behavior. Table I summarizes key notation from this formalization.

A program P is made up of a set of functions and interrupt handlers F with individual functions denoted as f. The set of functions called by f is denoted as C_f. The body of a function f is represented by a control flow graph $G_f = \{N_f, E_f\}$ with the set of nodes N_f consisting of program statements without control flow and a connectivity matrix $E_f = [e_{ij_f}]$ where e_{ij_f} is set to 1 if there is an edge from node i to node j in f and 0 otherwise. The number of successors, or the out degree, of a node $n_{i_f} \in N_f$ within a function f is $\text{out}(n_{i_f}) = \sum_{j \in N_f} e_{ij_f}$.

An ROI $R \subseteq F$ is a set of functions that the programmer is interested in tracing at runtime. Entry functions $R_e \subseteq R$ are the subset of functions from R that can be called by some $f \notin R$. Body functions $R_b \subseteq R$ are the subset of functions from R that can not be called by some $f \notin R$. For a given ROI, C_{f_R} is the set of functions called by f within the ROI.

The runtime call behavior $B = [b_{ij}]$ is a matrix where b_{ij} is the frequency with which function i calls j at runtime. Trivially, b_{ij} is zero if $j \notin C_i$. The runtime control flow is a set of matrices CFG. For function f, the matrix $CFG_f = [cfg_{ij_f}]$ describes the frequency with which each branch e_{ij_f} is traversed at runtime.

1) Global Identifier Logging: Global identifier logging is the commonly used mechanism for gathering call traces. This technique augments the header of each function within R to record a globally unique identifier when called and to record a return token when the function returns. It is significant to note that both the entry and body tokens occupy the same name space with global identifier logging.

2) Local Identifier Logging: Local identifier logging assigns token values from caller specific name spaces for each function within R_b or, should the caller be outside of R_b, from a name space spanning R_e. This is beneficial since only a small subset of functions within R are reachable from within a typical function, reducing the number of bits required to describe body token identifiers. Resolving such a token when reconstructing a call trace requires knowing the scope from which the token originated. This is facilitated by the parser that notes the last observed entry token and tracks the local scope as new body calls and returns are encountered.

3) Control Flow Logging: Control flow logging records detailed traces of runtime control flow decisions, rather than local call identifiers, within R_b. Entry calls to R and all returns from functions within R are handled in the same manner as local identifier logging. This provides the information necessary to reconstruct call traces. Segments of code without control flow require no logging, since functions along such a path are known at compile time to be deterministically called.

A. Discussion

Table II presents the cost for each logging technique in a standard form that highlights the bandwidth contributed by each of entry, body, and return tokens. In all logging strategies, returns from functions in the ROI must be logged to allow unambiguous call graph reconstruction. This results in the same return token cost in all three call tracing approaches. In this analysis we assume that a 1 bit return token is used.

For comparison of logging schemes, Table II uses the width function that computes the width in bits of an identifier. For this formalization we assume a fixed bit width encoding scheme that returns the minimum number of bits required to give each element in a size n set a unique identifier:

$$\text{width}(n) = \begin{cases} \lceil \log_2(n) \rceil & \text{if } n > 1 \\ 1 & \text{if } n \leq 1 \end{cases}$$

In practice the width function may use external information or heuristics to assign variable width identifiers to the set of tokens in a single name space. As an example of this, the evaluation in Section IV examines Huffman coding that uses variable width tokens optimally assigned from a priori call frequency information.

1) Comparing Global and Local Identifier Logging: From Table II we see that global and local identifier logging techniques generate the same number of entry tokens and the same number of body tokens, only differing in the size of identifier used. Entry and body token sizes in global identifier logging is determined by the number of functions in R. Entry and body token sizes in local identifier logging are typically smaller than the equivalent global identifier logging tokens, since the name spaces are smaller with $R_e \subseteq R$ and, for each $f \in R$, $C_{f_R} \subseteq R$. The greatest bandwidth savings typically come from the reduction in body token widths, since C_{f_R} is typically a much smaller name space than R.

2) Comparing Local Identifier and Control Flow Logging: Table II shows that the difference in bandwidth cost between local identifier logging and control flow logging is from the body token costs. Where the net body token cost for local identifier logging is dependent on runtime calls, the net body token cost for control flow logging is dependent on runtime control flow. In practice control flow logging requires recording more body tokens than local identifier logging, but these control flow tokens tend to be very short.

A fascinating distinction between control flow logging and the other call trace gathering techniques is the resolution of the resulting trace. With local or global identifier logging, the resolution of the trace is that of entry into and returns from ROI functions. With control flow logging, the resolution of the traces is that of the control flow within and returns from ROI functions.

3) Optimizing Control Flow Logging: Traces of control flow decisions through the ROI, combined with entry and return tokens, provide more than enough information to reconstruct a runtime call trace. Not logging control flow decisions that have no effect on call trace reconstruction provides significant bandwidth savings.

Our implementation of control flow call tracing evaluated in Section IV uses this optimization by performing a path-insensitive

TABLE II
BANDWIDTH CONSUMED BY THREE CALL TRACE LOGGING TECHNIQUES.

Total Cost	=	**Entry Token Cost**	+	**Body Token Cost**	+	**Return Token Cost**
$\mathrm{cost}(global)$	=	$\mathrm{width}(\lvert R_e\rvert + \lvert R_b\rvert) * \sum_{f\in F,\, g\in R_e} b_{fg}$	+	$\mathrm{width}(\lvert R_e\rvert + \lvert R_b\rvert) * \sum_{f\in F,\, g\in R_b} b_{fg}$	+	$\sum_{f\in F,\, g\in R} b_{fg}$
$\mathrm{cost}(local)$	=	$\mathrm{width}(\lvert R_e\rvert) * \sum_{f\in F,\, g\in R_e} b_{fg}$	+	$\sum_{f,\, g\in R_b} (\mathrm{width}(\lvert C_{f_R}\rvert) * b_{fg})$	+	$\sum_{f\in F,\, g\in R} b_{fg}$
$\mathrm{cost}(cfg)$	=	$\mathrm{width}(\lvert R_e\rvert) * \sum_{f\in F,\, g\in R_e} b_{fg}$	+	$\sum_{f\in R_b} \sum_{i,\, j\in N_f} (\mathrm{width}(\mathrm{out}(n_{i_f})) * cfg_{ij_f})$	+	$\sum_{f\in F,\, g\in R} b_{fg}$

context-insensitive reverse dataflow analysis that finds the reduced set of control flow decisions to log that still allows complete call trace reconstruction.

The value domain of the dataflow is a partial order over ordered sets of called body functions where for two such sets, S_a and S_b, S_a is less than S_b iff S_a is a suffix of S_b. An additional “must log” value is included in the value domain and is the maximum value in the partial ordering. The height of this domain is bounded by making any set of called functions containing more than a threshold number of calls equivalent to the “must log” state.

The join function examines each incoming ordered set of function calls. If all incoming ordered sets are the same, the join function sets the in-state to be that ordered set since all branches at this point in the program execute the same set of instructions in the same order and thus the control flow decision does not effect the resulting call trace. If any one of the sets is different, the join function elevates the in-state of the block to the special “must log” state. The “must log” state indicates that control flow decisions at that location in the program must be logged at runtime.

If the in-state is “must log”, then the transfer function sets the out-state of a block to be the ordered set of body functions that are called within the block during execution. Otherwise, the out-state is set to the concatenation of the ordered set of called body functions within the block and the in-state of the block.

IV. EVALUATION

Global identifier logging is implemented by recording a globally unique function identifier upon entry to an ROI function. Local identifier logging is implemented by recording an identifier, unique within the entry token space, upon entry to the ROI by an entry call and using locally scoped identifiers recorded at the call site for body calls within the ROI. Control flow logging is implemented by recording an identifier, unique within the entry token space, upon entry to the ROI by an entry call and using the dataflow analysis described in section III to drive placement of logs describing the branch taken at key control flow points. All three techniques log a short return token when any function within the ROI returns. Any function within the ROI that is also the target of a function pointer is conservatively treated as an entry call.

A. Results

Evaluation of the three call trace logging mechanisms was performed using the Avrora cycle accurate simulator [5] to simulate a small network of Mica2 sensor nodes measuring ambient light every four seconds and routing the sensed data to a single collector. A small testbed using physical Mica2 motes was used to validate the simulated results provided by Avrora, and observed log bandwidths consistently deviated by less than 0.5% from the simulated results when log bandwidth from dropped radio packets is accounted for.

Our evaluation examines the bandwidth required to generate logs using the three call trace logging mechanisms. We directly measured the bandwidth consumed using fixed bit width token encoding for each of the three call trace logging mechanisms. We then used detailed call traces from Avrora to re-encoded the tokens using

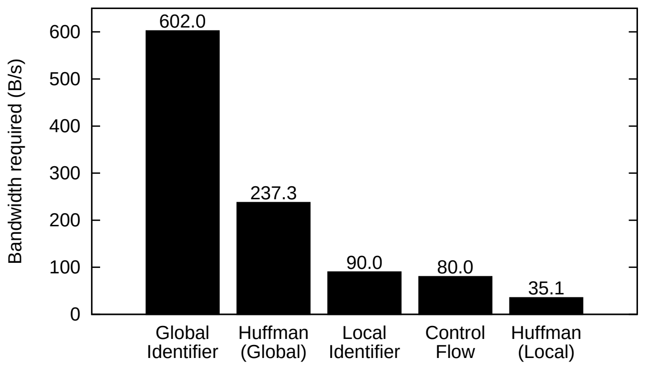

Fig. 1. Bandwidth consumed by various token encoding and call trace gathering techniques.

optimally sized variable width identifiers obtained using Huffman coding. We did not examine the frequencies of individual control flow decision points, so Huffman coding was not applied to control flow logging.

Figure 1 shows the log bandwidth consumed by the three logging systems using fixed bit width and Huffman coding with an ROI spanning a wide range of kernel, driver, and application subsystems. Both local identifier logging and control flow logging provided significant savings, consuming only 15% and 14% respectively, of the bandwidth of global identifier logging. Further, both local identifier logging and control flow logging also provided significant savings over Huffman coding of tokens from a global name space, further demonstrating the importance of dividing name spaces into smaller local scopes. Applying Huffman coding to local name spaces provides some additional gains over local identifier logging and control flow logging. However, any use of Huffman coding does require call frequency information that may be costly to obtain.

All of the logging techniques insert calls to a logging framework, resulting in an increase in the size of the program text segment. Global identifier logging provides a baseline measure of this impact, increasing the program text segment size of the ROI by an average of 31%. Local identifier logging has a greater impact and increases the ROI code size on average by 43%, since multiple caller side logging statements for a single target function are used rather than the single callee side logging statement used by global identifier logging. Insertion of logging calls at multiple control flow points results in control flow logging having the greatest impact on the code size of the ROI with an average increase of 66%. All of the logging techniques use the same underlying logging library that results in a fixed program text size increase and fixed increase in RAM utilization. Other than the RAM used by statically allocated buffers within the underlying logging library, the three logging techniques introduce no additional RAM overhead.

B. Discussion

The evaluated ROI was quite large to stress the alternate logging techniques, but in practice smaller ROI may be used by developers. While smaller ROI would reduce the spread in bandwidth consumption between global identifier logging and the other call trace gathering techniques, the analytical model shows that local identifier logging will never require more log bandwidth than global identifier logging. Smaller ROI also reduce the increase in program text segment size caused by instrumentation. The text size increase resulting from local identifier logging relative to that from global identifier logging is moderately stable across different ROI, while that from control flow logging is more sensitive to the structure of the instrumented ROI.

Logging a token introduces delay into the execution of the underlying system. All three logging techniques use the same logging library, so the delay introduced by logging a given token is equivalent for each of the three logging approaches. The global and local identifier logging techniques log two tokens for each traced function call and result in a small well defined delay. Delay introduced from control flow logging is more variable, since the number of tokens logged is dependent on the underlying control flow structure of a program and the actual path taken through the program at runtime. While no timing problems were observed in the evaluated ROI, interactions between real time code and delays resulting from calling into the logging library was formally not evaluated.

Our evaluation used a simple instrumentation strategy to evaluate the logging techniques, while more advanced implementations could further reduce overheads. For example, local identifier logging was accomplished by inserting logging statements into caller code directly before body calls. Alternate implementations could use runtime stack disassembly to infer the caller and then log a caller specific identifier from the callee. Such an approach only instruments the callee and thus reduces text size expansion, but at the cost of added implementation complexity and reduced implementation portability.

Both local identifier logs and control flow logs are made more compact by exploiting knowledge of the current execution context. This dependence on execution context could magnify the impact of log corruption from a faulty program or from missing data, since the log parser must search for a new context when resuming parsing after the faulty region of log data. Minimizing this loss is important, since the log is itself a debugging aid. Our current log parser searches for a new synchronization point when resuming parsing after encountering log corruption or after missing data, and drops average of 10 bits of data during the synchronization process. As with all logging systems, protecting log data using techniques such as software fault isolation is important to minimize the damage caused by faulty software.

Our analysis examines uniform bit width token encoding and optimal Huffman coding that requires call frequency information, but no other coding algorithms. Developing runtime adaptive coding mechanisms to allow the token encoding within a name space to evolve over time, as call frequency becomes available or as the behavior of a deployed system changes, would be an interesting extension to this work.

V. Conclusion

Our analysis and experimentation demonstrates the significant savings achievable through the use of the proposed local identifier and control flow trace gathering schemes. For general logging tasks we recommend using local identifier logging, which balances compact log bandwidth with only slight text size increases over global identifier logging. Control flow logging is an intriguing technique that has very different performance properties than global or local identifier logging due to its dependence on runtime control flow decisions. Due to the added complexity required to use control flow logging and its more substantial increase in text segment utilization, we recommend reserving it for more specialized environments with unique control flow properties that would maximize its effectiveness. Huffman coding can be used to further reduce log bandwidth when call frequency data is available, although the initial savings from reducing name spaces using local call or control flow logging is more significant than the reduction resulting from Huffman coding applied directly to a global token name space. The bandwidth savings that these new call trace gathering techniques allow logs to be collected that provide developers with the insight they need for answering questions about wireless embedded systems and provide analyses with richer data sets.

References

[1] K. Yaghmour and M. R. Dagenais, “Measuring and characterizing system behavior using kernel-level event logging,” in *USENIX*. USENIX, 2000.

[2] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, “Capturing, indexing, clustering, and retrieving system history,” *SIGOPS*, 2005.

[3] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, “Magpie: Online modelling and performance-aware systems,” in *HotOS*. USENIX, 2003.

[4] 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.

[5] B. L. Titzer, D. K. Lee, and J. Palsberg, “Avrora: scalable sensor network simulation with precise timing,” in *IPSN*. IEEE Press, 2005.

[6] P. Levis, N. Lee, M. Welsh, and D. Culler, “Tossim: accurate and scalable simulation of entire tinyos applications,” in *SenSys*. ACM Press, 2003.

[7] G. Werner-Allen, P. Swieskowski, and M. Welsh, “Motelab: a wireless sensor network testbed,” *IPSN*, April 2005.

[8] E. Ertin, A. Arora, R. Ramnath, V. Naik, S. Bapat, V. Kulathumani, M. Sridharan, H. Zhang, H. Cao, and M. Nesterenko, “Kansei: a testbed for sensing at scale,” in *IPSN*. ACM Press, 2006.

[9] 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 Press, 2008.

[10] G. Tolle and D. Culler, “Design of an application-cooperative management system for wireless sensor networks,” in *EWSN*. IEEE, 2005.

[11] 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.

[12] 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 Press, 2006.

[13] M. M. H. Khan, L. Luo, C. Huang, and T. F. Abdelzaher, “Snts: Sensor network troubleshooting suite,” in *DCSS*. Springer, 2007.

[14] V. Krunic, E. Trumpler, and R. Han, “Nodemd: diagnosing node-level faults in remote wireless sensor systems,” in *MobiSys*. ACM Press, 2007.

[15] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin, “Sympathy for the sensor network debugger,” in *SenSys*. ACM Press, 2005.

[16] M. Khan, , H. Le, H. Ahmadi, T. Abdelzaher, , and J. Han, “Dustminer: Troubleshooting interactive complexity bugs in sensor networks,” in *SenSys*. ACM Press, 2008.

[17] W. Archer, P. Levis, and J. Regehr, “Interface contracts for tinyos,” in *IPSN*. ACM Press, 2007.

[18] R. Shea, Y. Cho, and M. Srivastava, “Lis is more: Improved diagnostic logging in sensor networks with log instrumentation specifications,” UCLA, Tech. Rep. TR-UCLA-NESL-200906-01, 2009.