

Low Bandwidth Call Trace Logging for Sensor Networks

Roy Shea, Young Cho, and Mani Srivastava

University of Los Angeles, California, USA
roy@cs.ucla.edu, young@ee.ucla.edu, mani@cs.ucla.edu

Abstract. Call traces can provide detailed insight into the operation of distributed embedded systems. Developers inspect traces to understand and debug systems using manual and automatic techniques such as data mining. Correlation of traces between nodes provides a network level view of system. These traces are typically gathered by logging a globally unique identifier for each called function. Unfortunately, this naive call trace gathering technique results in excessive consumption of the limited memory, bandwidth, and energy resources available in wireless sensor networks.

This paper proposes three new call trace gathering techniques that are designed specifically for the computing platforms with extreme resource constraints. The first technique uses local name spaces and caller side logging to significantly reduce the bit size of function identifiers. The second technique reconstructs call traces from a log of the runtime control flow decisions made by a program. The third technique performs a novel reduction over a program's control flow graph to limit logging to control flow nodes effecting runtime call decisions. Our work automates the insertion of logging statements into source code for all the techniques described above.

Our experimental results show promising outlook where two of the techniques reduced the size of the log to less than 15% of traces produced by traditional methods. These savings make the new call trace capturing techniques attractive additions to the toolbox employed by developers and users of wireless sensor networks.

1 Introduction

Event traces are emerging as a powerful technique that allow developers and system maintainers to peer into the too often obscured operation of sensor networks [1–7]. Function call traces are one of the most flexible and useful form of event traces. Function call traces are intuitive to developers who already divide functionality in their program into individual functions. Such traces are already established in debugging through the ubiquitous appearance of function back traces in debuggers. More complicated event abstractions can be captured by call traces. Examples include: interrupts handlers implemented as an asynchronous function calls, network message send events triggered by a corresponding network send functions, and context switches controlled by a kernel scheduler function.

Sensor network and general system tools that gather function call traces already exist. These tools tend to focus on trace processing and not the efficiency of call trace generation. This paper acknowledges the value of call traces and moves forward with a new question. How can function call traces best be gathered in sensor networks systems?

Implementations for gathering system call traces that we've encountered instrument functions to log a globally unique identifier (ID) when the function is called. This technique is simple to implement and easy to understand, but wasteful in its creation of excessively verbose logs.

This paper attacks the waste created by unique function ID logging with a single observation: from a given point in a program only a subset of functions may next be called. This observation is critical because the overhead from call trace logging is directly related to the bit width of logged IDs. Globally unique IDs require $\log_2(n)$ bits, where n is the number of functions in the program. Fewer bits can be used if a smaller set of functions can be considered.

We begin reducing call trace log sizes by using functionally scoped identifier name spaces. Only a set of functions are reachable from a given function. Using this reduced set to generate an ID unique to the caller provides immediate gains over the global identifier approach. In a sense we transition away from using a single "global dictionary" requiring a greater number of bits to encode unique function IDs. Instead we propose using multiple "locally scoped" dictionaries that are each customized to contain only the functions reachable from within a given function. These locally scoped dictionaries create more compact identifiers. The first contribution of this paper is the description and evaluation of logging via local name spaces.

We continue by pushing the initial observation, that only a subset of functions are reachable from a given program point, to an extreme using "statement level" dictionaries. Where locally scoped dictionaries consider only functions reachable within a given function, statement level dictionaries consider only functions immediately reachable from a given program statement. A statement level dictionary needs at most a single entry for non-control flow statements since at most one subsequent call may be reached (ignoring interrupts, an issue described in detail in section 3). Control flow statements with n branches require at most n entries, one entry to describe the first call along each branch. In this extreme view the logged data becomes the path taken through the control flow graph (CFG) of a program. Function call traces can then be inferred from the recorded control flow path. The second contribution of this paper is a discussion and evaluation of inferring call traces from runtime CFG path logs.

CFG path logging overshoots the original goal of capturing function call traces. CFG path logs provide significantly more information than simple function call traces. Unfortunately, this additional information comes at a log size increase that reduces the benefits of using extremely concise statement level dictionaries. The third contribution of this paper is the description and evaluation of an algorithm to reduce a program's CFG without hindering function

call trace inference. The reduction prevents logging at CFG nodes that don't directly effect subsequent function calls.

In summary this paper describes in detail three call trace gathering techniques: using local functionally scoped identifiers, using logs of runtime CFG decisions, and using reduced CFG. The call trace gathering techniques are compared to each other and to the baseline obtained from global ID logging. This evaluation guides developers in adding efficient call trace logging frameworks to their systems.

Emerging tools to provide insight into the workings of sensor networks and more traditional systems are described in section 2. Section 3 describes in detail the implementations of the trace logging technique evaluated by this paper. Extensive evaluation in section 4 reveals how each of the different technique performs on wireless sensor network applications. Final comments and directions for future work conclude the paper in section 5.

2 Related Works

Understanding and debugging wireless sensor networks is both challenging and immediately beneficial to the sensor network community. A steady flow of research has helped to apply general techniques to the sensor network domain and to develop new techniques specifically for wireless and embedded sensor systems.

Many problems within sensor network applications are preventable. Harbor [8] provides memory safety using runtime software fault isolation techniques created for the embedded sensing domain. Safe TinyOS [9] uses CCured [10] and additional static analysis to provide type and memory safety in TinyOS programs. UTOS [11] extends this work by providing a sandboxing infrastructure that isolates system resources from untrusted software extensions. Our work benefits from any tool that prevents bugs from damaging a system. Such tools provide a trusted base upon which to gather call trace logs, and eliminate classes of problems that need not be looked for within the trace logs.

Tools such as Marionette [12] and Clairvoyant [13] provide interactive debugging interfaces. Marionette uses RPC to provide remote read and write access to program symbols in a deployed sensor network. This facilitates rapid server side scripting of diagnostic programs monitoring system state. Clairvoyant extends these ideas by providing a debugging interface to remote nodes. Clairvoyant supports traditional debugging techniques such as break points and variable watches points. Clairvoyant also introduces new debugging abstractions specifically for sensor networks such as network wide break points. These debugging interfaces expect an interactive network debugging model. Function call traces presented in this work provide a higher level perspective of a system's operation. We envision using this higher level understanding gained from call trace logs to focus subsequent use of interactive debugging tools.

An alternate to interactive debugging is the embedding of checkpoint code directly within applications. One approach to this uses interface contracts [14] to interpose programmatic contracts between component interfaces. This allows

the runtime system to monitor the input and output of interfaces and act upon contract violations. Declarative tracepoints [15] provides a declarative scripting language that developers can use to encode arbitrary program monitors. Binary rewriting allows these scripts to be inserted and removed from a deployment at runtime. These works provide tools a developer can use to perform arbitrary monitoring of system internals at runtime. This is in contrast to call trace monitoring that provides only a single form of runtime monitoring, but that no effort from system designers. As with the interactive debugging techniques, we propose using call trace logging to direct these more concentrated but labor intensive monitoring operations.

Understanding the inner operation of a sensor network system is important for debugging, optimizing, and learning about deployed systems. A good deal of work in the sensor network community focuses on this task. Simulation allows nearly limitless visibility into a system. TOSSim [16] provides a nice application level simulation of sensor networks by compiling mote code for PC architectures. This facilitates testing protocol level correctness of applications on a single computer. Hardware simulators such as the cycle accurate Avrora [17] simulator allow lower layer bugs dependent on underlying hardware to be revealed. Extensive logs, including detailed call traces, can be obtained from these simulations. But the real world is a complicated beast and problems consistently emerge in sensor networks even after simulated testing. The optimized call traces logging techniques described in this paper are able to provide valuable insight into a deployed system that has matures out of the simulation stage.

An intermediate step between simulation and real world deployments are sensor network testbeds. Both Motelab [18] and Kansei [19] are examples of testbeds providing hardware back channels to large collections of physically deployed nodes. Such a test bed is valuable in its ability to test many nodes in a real environment that simulation fails to completely model. Extensive infrastructure provides capable back channels to each node. Test beds can use the techniques described in this paper to automate call traces gathering. Our research also extends to the final deployment stage when tethers are severed and a deployment goes live in the wild. In such an environment the optimizations we present are critical for operation on the resource constrained embedded systems.

The sensor network troubleshoot tool kit (SNTS) [4] provides an extended form of logging for deployments with minimal impact on the sensing system. SNTS focuses on diagnosing faults from distributed component interactions. Collector nodes operating independently from the main deployment are strategically placed within the network to passively record messages between nodes. After a data collection phase the logs from collector nodes are gathered at a backend for analysis and problem inference. SNTS minimizes the impact on deployed network by using separate dedicated logging hardware. Separate hardware allows SNTS to perform aggressive logging on externally visible events, such as writing all observed network traffic directly to external flash. Call trace logging collects state not normally exposed outside of a node. This forces call trace logging techniques on the same hardware as the system being monitored. This motivates developing

a logging infrastructure that consumes minimal resources on the host devices to minimize impact on monitored nodes.

Call trace logging is a representative logging technique. Logging tools specifically designed for sensor networks are emerging as the field matures. SNMS [1] is an early tool focused on logging in sensor networks. SNMS was designed to diagnose problems in live deployments. SNMS combines a logging infrastructure with a dedicated radio stack to increase access to diagnostic data in deployments. SNMS provides a framework for writing state marshaling routines to monitor symbols of interest. SNMS also extends the traditional `printf` style logging to use compile time compacted encodings of programmer specified debug strings. A query mechanism provides one shot or periodic access to any of the annotated program states. SNMS requires significant interaction to annotate variables of interest and write marshaling routines. Call traces gathered in an efficient manner can provide good initial insight into a sensor network. Once the developer has gained a better idea of where a problem exists or a subsystem that they want to know more about from the call trace logs, they can switch to tools such as SNMS that allow them to embed more detailed trace requests in the code and interactively query system state.

NodeMD [5] provides a fault detection, notification, and diagnostic system. The diagnostic subsystem keeps a circular buffer recording key system events such as interrupt firings, timer firings, and thread actions. Rather than recording the source of an event, the paper argues that recording only what type of event occurred is sufficient since most parts of a system have a unique event generation fingerprints. NodeMD and our call trace logging work strive to achieve extremely compacted logs. The call trace logging frameworks described in this paper focuses on a narrower domain than NodeMD. This decision allows our call trace logs to provide more details about subset of call events, such as what function is being called, at the cost of not explicitly logging other event types.

Both SNMS and NodeMD provide diagnostic modes to facilitate diagnostic interaction with a failed node. We see great value in building upon the diagnostic mode of SNMS and NodeMD and have found no technological reasons inhibiting generic call trace logging mechanisms from doing so.

EnviroLog [3] records expressive event traces on individual sensing devices. Developers use a simple API provided by EnviroLog to annotate modules, functions, and variables for logging. EnviroLog can then replay the traces in the otherwise live and deployed network, facilitating tuning and repeatable evaluation of wireless sensing systems. EnviroLog is best tuned for recording low frequency (about 10 Hz) events, such as sensor inputs, for long periods of time for use in replay. Call trace logs must record function calls that can occur at significantly higher frequencies upwards of 1000 Hz.

A good deal of prior work focuses on what to log and when to log it. PAD [6] provides a nice alternative to this research style by exploring extremely low overhead methods of deriving neighbor connectivity graphs in sensor networks and using this data to probabilistically diagnose root causes of problems. PAD is similar to our work in its concentration on finding better ways to log a single

key piece of data. Where PAD focuses on network connectivity, our work focuses on function call traces.

Logs and diagnostic data gathered from networks are of no value unless they are used. Some recent research explores uses for this data. Sympathy [2] uses periodic heart beats to gather connectivity, network flow, and node level statistics from a network. A base station monitors these heartbeats and uses decision trees to diagnose errors within the network. Recent work in the Dustminer project [7] combines event logging similar to EnviroLog [3] with an extending form of distributed pattern data mining seen in SNTS [4]. These styles of analysis can be applied to the function call traces gathered by the techniques proposed in this work.

A number of extensive logging tools are available for server and desktop class systems. Tools such as the Linux Tracing Toolkit (LTT) [20] and the work of Cohen et. al. [21] provide extensive insight into single systems. Magpie [22] and Pip [23] provide similar visibility into distributed systems. All of these projects help demonstrate the value of extensive system logging. However, the extensive resource used by all these projects limit their applicability to the wireless embedded devices.

We view the work presented in this paper as an application specific compression technique for call traces in the wireless embedded systems domain. An alternative to the techniques evaluated in this paper is the use of traditional lossless data compression to reduce the size of call traces. Prior work [24] indicates that traditional compression may require hardware aware optimizations for gains to be realized on embedded systems. Other forms of lossless compression are simply not suited for operation on the lowest end embedded processors used in sensor networks.

3 Logging Function Call Traces

Our goal is to produce detailed traces that give developers and users insight to the execution within their wireless and embedded sensing systems. This work provides that knowledge by creating logs that capture the run time call traces of bottom tier wireless sensing devices.

There exist many sources of data for constructing function call traces. This paper evaluates four techniques that are able to supply such data. We explore two classes of techniques for gathering call traces: (1) recording identifiers (IDs) for called functions and (2) recording control flow graph (CFG) information to infer called functions. The first technique is optimized by choosing an identifier name space that minimizes the number of IDs. Control flow graph records can be reduced in size by only recording CFG decisions of interest.

3.1 Recording Function Identifiers

The simplest of our evaluated techniques uses global function identifiers. Global function identifiers are trivially gathered by adding a brief logging preamble at

Listing 1.1. Light sensor handling routine.

```
1 int handle_ligth_data(uint8_t light_intensity) {
2
3     static uint16_t dark_count = 0;
4     static uint16_t average_light = 0;
5     uint16_t delta;
6
7     delta = abs(average_light - light_intensity);
8
9     if (delta > DETECTION_THRESHOLD) {
10         broadcast_detection_event(light_intensity, NODEID);
11         average_light = reset_average(light_intensity);
12     }
13     else
14         average_light = ewa(average_light, light_intensity);
15
16     if (light_intensity < 10)
17         ++dark_count;
18
19     return dark_count;
20 }
```

the top of each function implementation. This preamble logs the unique ID of the called function. Unique function IDs are encoded using $\log_2(\text{num_functions})$ bits. The simplicity of this technique is its greatest merit and probably the reason that the technique is currently used. Our evaluation uses global IDs as a baseline.

The primary overhead of trace logging is the size of the IDs used to encode traced functions. If the program containing the `handle_light_data` routine in listing 1.1 contained 200 functions, then eight bits of data would be required for each globally unique ID and the function call on line seven would consume eight bits of log space. But from within a given function, only a small subset of functions are reachable. Only four functions are called from within `handle_light_data` so two bits of information can differentiate each function. By inserting ID logging into the caller code, directly before a function call is made, we can use this reduced name space and enjoy more compact function IDs.

Complications also arise from the unpredictable changes in program flow caused by function pointers and hardware interrupts. Special symbols are added to the tracing system to accommodate these jumps by recording the jump action, unique target identifier, and return from such a jump.

This modification significantly reduces the trace bandwidth overhead, but requires more invasive code modification and higher code text size inflation. Local name spaces require a logging statement at each call site withing a program. This is the second technique that we implement and evaluate.

3.2 Recording Runtime Control Flow Graph Decisions

The purpose of call trace logging is to provide end users with a detailed view of run time program flow. The techniques described above provide this information by logging the ID of each function called. It is trivial to reconstruct the call trace

using this information. Unfortunately, even with local name spaces much of the information provided within such a log is redundant. Local name spaces remove excess data from logs by choosing function IDs from a reduced name space. The control flow graph (CFG) logging techniques take this idea to its extreme by forming a name space for every point in the program.

For example, upon entry to the `handle_light_data` routine in listing 1.1 the `abs` function must be called next. We thus create a reduced name space for lines 1 - 7 with `abs` as its only member. Line 9 will use a different name space containing two entries, `broadcast_detection_event` and `ewa`, corresponding to the function call at either line 10 or 14 resulting from the alternate branches of the conditional.

In this extreme the name spaces become an encoding of the path taken through the CFG of a program at runtime. Segments of code without control flow require no logging since functions on the path are known at compile time to be deterministically called. Logging statements are inserted at the top of branch point successors statements to record the branch that is taken at runtime.

A number of details must be handled in this approach. Path data is relative. Its utility depends on knowing where within the system execution is at any given time. When a starting point is known the path encoding mechanism described above can be used to uniquely identify each function called. As with local name space ID recording, complications arise from the unpredictable changes in program flow caused by function pointers and hardware interrupts. Special symbols are added to the tracing system to accommodate these jumps by recording the jump action, unique target identifier, and return from such jumps. A final ambiguity arises from deterministic sequences of calls, such as lines 10 - 11 in listing 1.1. While it is clear that `broadcast_detection_event` will be called followed by `reset_average`, it is not immediately clear from the CFG data when the first function returns and the second is called. Consequently, we add an additional token to CFG traces to explicitly record function returns. This is the third technique that we implement and evaluate.

The logs collected by recording the runtime CFG traversal described above overshoots the original stated goal of collecting function call traces. A CFG trace of listing 1.1 will note which branch is taken after the conditional at line 16. But this decision has no direct effect on the functions that called at runtime. CFG path logging can be optimized for call trace inference by only logging control flow decisions that directly effect called functions. Recording control flow decisions from a reduced CFG path is the fourth technique that we implement and evaluate. The following is a high level description of the algorithm we use to reduce the CFG to a subset of nodes that must be recorded to reconstruct call traces.

1. Nodes with no outgoing edges in the original control flow graph must be a return statement. These can be safely ignored.
2. Nodes with one outgoing edge in the original control flow graph are a sequence of instructions where:

- (a) The sequence of instructions does not include any calls and this node has no descendants requiring trace information recording. This node is ignored.
 - (b) The sequence of instructions does not include any calls, but this is a parent to a child node requiring trace information recording. This requirement is passed up the tree and this node is ignored.
 - (c) The sequence of instructions includes at least one call. This node requires trace information recording the path to the node. This node consumes the state of children nodes and the new requirement is passed up the tree.
3. Nodes with two or more outgoing edges where:
- (a) One or more children nodes loop back within the control flow graph:
 - i. None of the children nodes require trace information record then:
 - A. If all children loop back to the current node, then all children and the current node may be ignored.
 - B. If all children loop back to the same target node, but not the current node, then all children may be ignored and it should be noted that the current node is part of a loop to the target node.
 - ii. If all children are members of loops bounded by the current node, then all children and the current loop may be ignored.
 - (b) Some of the children nodes require require trace information recording:
 - i. If all children require the same trace information, all children can be ignored and the trace information can be passed up through the current node.
 - ii. Else all children are included in the reduced graph.
 - (c) None of the children are members of loops and none of the children require trace information recording. All children may be ignored.

3.3 Token Encoding

Each of the techniques described above writes tokens to form a log. Global name spaces use a single token set consisting of a unique ID token for each function. Local name spaces use a token set for each functionally scoped name space, with tokens encoding the locally reachable function IDs. Control flow logging requires tokens to describe: the branch taken, function pointer and interrupt dispatches, function pointer and interrupt targets, and function returns.

For both ID recording techniques we use a uniform $\log_2(n)$ bit-width to encode ID tokens, where n is the number of tokens in the set. Dedicated jump and return tokens are added for the local name space technique.

The four tokens used by the CFG techniques are 0, 1, **return**, and **jump**. The branch taken by a conditional is encoded using a string of 0s and 1s with length $\log_2(n)$, where n is the number of branch destinations. We decided to use two different token encoding schemes after manual inspection of the CFG tokens generated by different sensor network subsystems. We observed that subsystems making many function calls tended to generate more 0 and 1 branch tokens. In

contrast were subsystems making very few function calls, that tended to implement accessor and helper functions that generated mostly `return` tokens. Based on these observations we implemented a simple static classifier that examines the ratio of function calls to declared functions in a subsystem. Subsystems with a high ratio of function calls to declared functions use 1, 2, 3, and 3 bits to encode each of the 0, 1, `return`, and `jump` tokens. This encoding favors code with more complex control flow. Subsystems with a low ratio of function calls to declared functions use 2, 3, 1, and 3 bits to encode each of the 0, 1, `return`, and `jump` tokens. This encoding favors code with simple control flow.

Different tokens in each technique will occur with different frequencies. When system profiling is able to provide token frequencies, further optimization of all techniques can be performed by using a custom encoding scheme to minimize the size of logs generated by a token set. We leave this form of adaptive token encoding as future work.

4 Evaluation

We evaluate the suitability of the four call trace gathering techniques described in section 3 for application in distributed embedded systems, such as sensor networks, that have access to only limited resources. Our evaluation begins with a more detailed look at how each of the call trace gathering techniques was evaluated, before providing a detailed examination of resulting log size. The evaluation continues with a look at overheads introduced by each tracing technique, since excessive overheads would devalue log size gains.

4.1 Instrumentation Framework

We implemented a utility for each call trace logging technique to automate the insertion of logging statements into source files. Instrumentation is currently performed one source file at a time, allowing system designers to easily instrument only subsystems of interest that are then compiled into the deployed program image. Call trace instrumentation is implemented using a combination of CIL [25] and Python.

The call tracing techniques described in this paper are agnostic to the specific log management, long term storage and transport facilities provided by other parts of the system. Since this work emphasizes reducing the size of call traces, our evaluation does not focus on issues such as log storage or log routing. Such topics are interesting areas for research in their own right and discussed elsewhere [1, 5, 6]. For completeness we briefly highlight the underlying system components that the evaluation rests upon.

Call traces are simply bit streams. In this evaluation the call trace bit stream is fed to a standalone logging component that records bit streams in 32 byte buffers. Full 32 byte buffers are wrapped in a header with a one byte log size (required when forcing a flush of partially full logs), one byte log type, and one byte log sequence number. The encapsulated log buffer is then sent via a simple

routing tree to a single collection sink. Each node in the routing tree chooses a best “next hop” based on active beacons sent periodically sent by neighboring nodes.

Logs received at the backend are passed to a log parser that gathers per-node log statistics, tokenizes logs, and decodes the call traces. Decoded call traces can be sent directly on to external tools for data mining [7, 4] and other analysis [2], or displayed in a simple graphical interface for manual inspection by system developers and maintainers. All of the call trace gathering techniques are robust to lost or corrupted data, should the log gathering layer not provide adequate error checking. Most sensitive to such corruption are techniques using CFG path tracing. The use of explicit IDs for dispatches via function pointers or interrupts act to quickly resynchronize the location in program code after such a problem occurs. When such an occasion occurs our backend notes the loss or corruption of data and resumes generating output after the next resynchronization point.

Most of the evaluation is performed using the Avrora [17] simulator to model a network of Mica2 sensor nodes, providing easily reconfigurable and controllable test environment. Our evaluation uses only the output from the simulated sink and the suite of call trace gathering tools designed for real deployments, allowing an identical setup to be used during cross verification on our testbed of Mica2 nodes. A Mica2 testbed is used to verify the observed simulation results.

4.2 Call Trace Log Size

The primary goal of this research is reducing the size required for capturing call trace logs. To understand the efficiency of each call tracing technique we compared the size of call traces from 37 source files, 31 from the operating system kernel and 6 from a data gathering application, using each technique.

We chose to limit our analysis to those the majority of system components that require no manual changes for instrumentation with logging. The only systems not evaluated were the low level interrupt handlers used by the radio stack, the main task scheduler, and memory buffer manager. The radio stack uses the SPI interrupt to drive the radio. The SPI runs at the speed of the radio, even when there is no radio traffic. The brief SPI interrupt handler, its call into the radio, and a subsequent call into the timing subsystem account for an exceptional volume of traffic that quickly overwhelms the logging infrastructure. Close interactions between the task scheduler and memory buffer manager would require additional infrastructure to be added outside of the call trace gathering system to prevent the act of logging from triggering a cascade of events that themselves require logging.

Table 1 shows the rate of log growth resulting from aggregating the 37 logs for each call tracing technique. Using global IDs to record call traces requires support for significantly faster growing, and thus larger, logs. Complete control flow graph traces can be collected using only 38% of the space required by naively logging global IDs. Local ID logging and reduced CFGs further reduce call trace requirements, both requiring about 15% of the space used by global ID call tracing.

Logging Technique	Aggregate Rate of Call Trace Log Generation (B/s)	Relative Rate of Call Trace Log Generation
Global IDs	602	100%
Local IDs	90	15.0%
Full CFG	229	38.1%
Reduced CFG	89	14.8%

Table 1. Rate of log growth for recording call traces using four different trace recording techniques.

It is very interesting that all CFG decisions can be logged in less space than that required by global ID logging. Full CFG traces provide a more detailed view of system operation than call traces by recording the actual path taken through each called function. Systems that currently use global ID traces can transition to full CFG logging to reduce log sizes and gain a good deal more information about system operation. Both local ID logging and reduced CFG logging provide call trace data using even smaller logs and may make call trace logging a more viable solution for systems that found global ID logging to incur too much overhead. Their average performance is too close to clearly state that one is better than the other without further investigation.

Logging Technique	Initial Rate of Call Trace Log Generation (B/s)	Subsequent Rate of Call Trace Log Generation (B/s)
Global IDs	605	601
Local IDs	87	91
Full CFG	236	228
Reduced CFG	98	87

Table 2. Rate of call trace log growth for the 30 seconds after rebooting and for the later steady state operation.

We continued our evaluation by comparing log generation during the first 30 seconds after rebooting a node, to logs generation from later steady state operation. We were concerned that the flurry of boot time activity could overload logging mechanisms, prompted the log parser noting that the initial log packets were almost always lost for one particular subsystem. Table 2 summarizes the findings. While none of the techniques observed substantial changes in log rate growth, both techniques based on CFG path recording observed higher rates after rebooting the nodes. Inspection of individual component logs revealed abnormally high call trace data emerging from a boot time linker used by the operating system and implemented with a deeply nested conditional loops. This finding does reveal the heightened sensitivity to program structure that the CFG based logging approaches display. Amusingly, this finding also confirms the omi-

nous comment found atop the troublesome block of code, “NOTE: it will be good idea to profile this. . .”

Logging Technique	Rate of Kernel Call Trace Log Generation (B/s)	Rate of Application Call Trace Log Generation (B/s)
Local IDs	72	18
Full CFG	199	30
Reduced CFG	63	26

Table 3. Rate of call trace log growth from kernel components and from application components.

Table 3 compares call trace log generation rates between components common to the operating system kernel and components from the application running on top of this kernel is also revealing. Call trace logging of application code using local IDs is on average more efficient than that using reduced CFG patch traces, while the opposite holds true for kernel call traces. Looking at the reduction achieved from full CFG traces, we can see that the CFG reduction algorithm is more efficient on kernel code than user code. Global ID tracking is not included in this comparison due to its use of caller, rather than callee, generated traces.

Subjective code inspection leads us to believe that this results from kernel assuming a more direct form with occasional tight loops effecting local data, both of which favor reduced CFG path logging. This is in contrast to application code that tends to have more complicated control flow triggering one of a few distinct actions, which favors local ID tracking. This result can help direct future users of call trace logging to use a technique best suited for the domain they wish to examine.

Rate of call trace generation from different components varies greatly. Many subsystems generated 20 or fewer bits of call trace data during during startup and none during steady state operation, with 10 of the 37 subsystems generating absolutely no call trace data in any of the evaluated systems.

To verify the validity of those results gained from simulation, we reran a subset of the evaluation on a testbed. The same tools and program images were used to examine the rate of call trace generation for the application layer components of a simple sensing application. These components implemented functionality for: photo sensor driver, neighborhood table maintenance, tree routing, and periodic sensor sampling. In all cases the simulated rate of call trace log generation deviated by less than 0.5% of the rate observed from the testbed.

4.3 System Overhead from Call Trace Logging

Call trace logging incurs system overhead including an increase in program text size, memory usage, and CPU overhead. Specific domain constraints, especially

available space for the program image, may mandate the use of one call tracing technique over another.

Logging Technique	Program Text Size (B)	Relative Size Increase	Size per Line of Source Code (B / line)
None	53556	0%	4.45
Global IDs	70046	31%	5.83
Local IDs	75810	42%	6.30
Full CFG	100720	88%	8.38
Reduced CFG	88932	66%	7.40

Table 4. Increase in program text size from insertion of call trace statements.

Insertion of call traces into an application results in an increased program text size. Table 4 shows text sizes. Local ID tracking and reduced CFG patch recording respectively cause 42% and 66% increases in text size. Note that these size increases are for enabling logging across 37 system components. The significance of these size increases depends on the underlying embedded hardware and application needs. For example, the ATmega128 used on Mica2 and MicaZ sensor nodes has a relatively spacious 128k of program flash that can often accommodate a larger program image. The MSP430 used in more recent sensor nodes has only 48k of flash that may require developers to be more judicious in what subsystems they wish to enable call trace logging within.

All of the evaluated systems have the same minimal RAM overhead. This overhead is caused by the buffering of trace data by the external log buffering component. The log buffering mechanism used in this evaluation requires 61 B of RAM for internal state and uses an additional 72 B of RAM for the pair of bit buffers it alternates writing data to. Note that this log buffering mechanism is external to the call trace gathering techniques we develop in this paper and may be exchanged for other log management components.

CPU overhead for logging is also relatively small. Profiling using Avrora reveals that the average call to log 3 bits of data requires 846 CPU cycles on the ATmega128 micro controller. Note that individual log calls can choose to log fewer or more bits at a single time, resulting in a slightly smaller or greater CPU overhead. More significant are the occasional buffer flushes triggered by filling a 32 byte buffer. The buffer flush consumes approximately 2000 additional cycles before posting the buffer to the radio. The worst case consumes approximately 3000 cycles, or about a third of a millisecond on an 8 MHz CPU. Any delay has the potential to cause problems in timing sensitive code, such as some interrupt handlers, and should be considered by system designers instrumenting timing sensitive code to gather detailed call traces. Fortunately, nearly all of the logging code may be preempted. Delays from logging did not cause any perceivable problems in the evaluated subsystems.

5 Conclusion

The evaluation helps to demonstrate the significant gains that can be made in reducing the size of logs generated by function call tracing. This minimization is important for reducing RAM and flash requirements for storing logs on nodes, and for minimizing bandwidth consumed when transporting logs through a network of devices.

An interesting design choice was our decision to not support parameter logging, which can provide even more insight into a system but generating substantially larger logs. Such functionality could be added on top of the current call trace logging by annotating parameters of interest with an attribute. Such parameters would then be inserted into the call trace log stream when encountered. However, we feel that the lower overhead associated function call traces makes them an important exploratory tool to use in systems, saving the higher overhead techniques for use after a basic idea about the problem has been formed. Exploring low overhead techniques for recording parameter and variable traces is an area that we are interested in pursuing.

The logging provided by full CFG tracing reduces the generating logs that are 38.1% smaller than those from global ID logging. Extensive use of logging statements leads to an 88% increase in program text size for programs with full CFG logging, compared to the 31% increase from global ID logging.

The two most promising call tracing techniques are recording locally scoped function identifiers and recording a reduced runtime CFG path. Local ID logging and reduced CFG logging provides logs that are only 15.0% and 14.8% of the size of logs generated by globally unique IDs, respectively. This reduction in log size generation comes at the cost of increased program text size. Where global ID logging increases text size by an average of 31%, local ID tracing increases average text size by 42% and CFG logging increases average text size by 66%.

We look forward to carrying these ideas further in our continued efforts to expose state currently hidden in deployed sensor networks. In an effort to foster open research and help the community build upon each others work, the source codes for tools described in this paper are available for download at <http://anonymous>.

References

1. Tolle, G., Culler, D.: Design of an application-cooperative management system for wireless sensor networks. In: EWSN, IEEE (2005)
2. Ramanathan, N., Chang, K., Kapur, R., Girod, L., Kohler, E., Estrin, D.: Sympathy for the sensor network debugger. In: SenSys, ACM (2005)
3. Luo, L., He, T., Zhou, G., Gu, L., Abdelzaher, T.F., Stankovic, J.A.: Achieving repeatability of asynchronous events in wireless sensor networks with envirolog. INFOCOM (2006)
4. Khan, M.M.H., Luo, L., Huang, C., Abdelzaher, T.F.: Snts: Sensor network troubleshooting suite. In: Distributed Computing in Sensor Systems, Springer (2007)

5. Kronic, V., Trumpler, E., Han, R.: Nodemd: diagnosing node-level faults in remote wireless sensor systems. In: *MobiSys*, ACM Press (2007)
6. Liu, K., Li, M., Liu, Y., Li, M., Guo, Z., , Hong, F.: Pad: Passive diagnosis for wireless sensor networks. In: *SenSys*, ACM Press (2008)
7. Khan, M., , Le, H., Ahmadi, H., Abdelzaher, T., , Han, J.: Dustminer: Troubleshooting interactive complexity bugs in sensor networks. In: *SenSys*, ACM Press (2008)
8. Kumar, R., Kohler, E., Srivastava, M.: Harbor: software-based memory protection for sensor nodes. In: *IPSN*, ACM Press (2007)
9. Coopriider, N., Archer, W., Eide, E., Gay, D., Regehr, J.: Efficient memory safety for tinyos. In: *SenSys*, ACM Press (2007)
10. Necula, G.C., Mcpeak, S., Weimer, W.: Ccured: Type-safe retrofitting of legacy code. In: *Symposium on Principles of Programming Languages*. (2002)
11. Regehr, J., Coopriider, N., Archer, W., Eide, E.: Memory safety and untrusted extensions for tinyos. Technical Report UUCS-06-007, University of Utah (2007)
12. Whitehouse, K., Tolle, G., Taneja, J., Sharp, C., Kim, S., Jeong, J., Hui, J., Dutta, P., Culler, D.: Marionette: using rpc for interactive development and debugging of wireless embedded networks. In: *IPSN*, ACM (2006)
13. Yang, J., Soffa, M.L., Selavo, L., Whitehouse, K.: Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. In: *SenSys*, ACM (2007)
14. Archer, W., Levis, P., Regehr, J.: Interface contracts for tinyos. In: *IPSN*, New York, NY, USA, ACM Press (2007) 158–165
15. Cao, Q., Abdelzaher, T., Stankovic, J., Whitehouse, K., Luo, L.: Declarative tracepoints: A programmable and application independent debugging system for wireless sensor networks. In: *SenSys*, ACM Press (2008)
16. Levis, P., Lee, N., Welsh, M., Culler, D.: Tossim: accurate and scalable simulation of entire tinyos applications. In: *SenSys*, ACM Press (2003)
17. Titzer, B.L., Lee, D.K., Palsberg, J.: Avrora: scalable sensor network simulation with precise timing. In: *IPSN*, IEEE Press (2005)
18. Werner-Allen, G., Swieskowski, P., Welsh, M.: Motelab: a wireless sensor network testbed. *Information Processing in Sensor Networks*, 2005. *IPSN 2005*. Fourth International Symposium on (April 2005)
19. Ertin, E., Arora, A., Ramnath, R., Naik, V., Bapat, S., Kulathumani, V., Sridharan, M., Zhang, H., Cao, H., Nesterenko, M.: Kansei: a testbed for sensing at scale. In: *IPSN*, ACM Press (2006)
20. Yaghmour, K., Dagenais, M.R.: Measuring and characterizing system behavior using kernel-level event logging. In: *USENIX*, USENIX (2000)
21. Cohen, I., Zhang, S., Goldszmidt, M., Symons, J., Kelly, T., Fox, A.: Capturing, indexing, clustering, and retrieving system history. *SIGOPS* (2005)
22. Barham, P., Isaacs, R., Mortier, R., Narayanan, D.: Magpie: Online modelling and performance-aware systems. In: *HotOS*, USENIX (2003)
23. Reynolds, P., Killian, C., Wiener, J.L., Mogul, J.C., Shah, M.A., Vahdat, A.: Pip: detecting the unexpected in distributed systems. In: *NSDI*, USENIX (2006)
24. Barr, K., Asanović, K.: Energy aware lossless data compression. In: *MobiSys*, ACM Press (2003)
25. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of C programs. In: *CC*, Springer (2002)