Demo Abstract: Diagnostic Tracing of Wireless Sensor Networks with TinyTracer

Vinaitheerthan Sundaram
School of Electrical and Computer Engineering
Purdue Univsersity
West Lafayette IN USA
vsundar@purdue.edu

Patrick Eugster, Xiangyu Zhang
Department of Computer Science
Purdue Univsersity
West Lafayette IN USA
peugster, xyzhang@purdue.edu

ABSTRACT

Run-time debugging tools are required to detect and diagnose post-deployment failures in wireless sensor networks. Reproducing a failure from the trace of past events can play a crucial role in diagnosis. We describe TinyTracer, an efficient interprocedural control-flow tracing tool that generates the trace of all interleaving concurrent events as well as the control-flow paths taken. TinyTracer enables reproducing failures at a later stage, allowing the programmer to diagnose failures effectively. In this demo, we demonstrate the ease of use of TinyTracer. We see TinyTracer as an important tool for post-deployment diagnosis, which can enable future research on trace-based debugging approaches for wireless sensor networks.

1. INTRODUCTION

Wireless Sensor Networks (WSNs) are being increasingly deployed in various scientific and industrial domains to understand the micro-behavior of physical phenomena. Unexpected deployment failures have been observed in many of the large WSN deployments despite thorough testing in the lab prior to the deployment [1]. Diagnosing these failures is critical to fix or take preventive measures to avoid same failures in the future.

We proposed TinyTracer [3], a novel efficient interprocedural control-flow tracing tool that generates the trace of all interleaving concurrent events as well as the control-flow paths taken. TinyTracer enables reproducing failures at a later stage, thus allowing the programmer to diagnose failures effectively. The control-flow information is effective for diagnosis and yet, can be captured in a resource-efficient way.

TinyTracer is easy to use because it automatically instruments the code to trace the components and functions of interest to the developer. The trace collected can be viewed using a trace parser that pretty prints the raw trace. We see TinyTracer as an important tool for post-deployment diagnosis which can enable future research on trace-based debugging approaches for WSNs. In this demo paper, we describe our technique briefly, followed by the implementation and the usage of our tool.

2. DESCRIPTION

TinyTracer is an efficient tracing technique that encodes and records the interleaving of all concurrent events as well as the control-flow paths taken. TinyTracer represents the interprocedural control-flow path inside an event as one or two byte integer. This is achieved by statically analyzing the program and automatically instrumenting only a few statements inside event handlers. The instrumentation per event consists in adding increment statements at every branch and two function calls to record the event identifier and the control-flow path taken.

At run-time, the interprocedural control-flow path taken is computed and recorded along with the event identifier in the RAM. The trace recorded in RAM is compressed before being committed to non-volatile external flash memory or sent in the radio.

Since the trace contains all the control-flow decisions, the trace can be replayed in a controlled environment such as a simulator or a debugger to reproduce the fault. In fact, the trace enables reverse debugging of the program, thus allowing the programmer to identify the fault effectively. Our tracing method satisfies the stringent resource constraints of WSNs, by exploiting the specific execution model and making use of compression techniques. While explained and implemented in the context of nesC/TinyOS, TinyTracer is generalizable and can be easily adapted to other environments.

3. IMPLEMENTATION AND USAGE

We implemented TinyTracer using open-source tools, namely, CIL [2], TinyOS and the nesC compiler. CIL is a source-to-source transformation tool for C from UC Berkeley. Our implementation has two core components: (1) the TinyTracer engine, a compile-time CIL module that does interprocedural analysis and *automatically* instruments the code. (2) `TinyTracerC`, a run-time nesC component that records the trace, compresses it and either commits it to the flash or sends it to the base station.

The compile-time workflow of TinyTracer is shown in Figure 1. To trace a TinyOS application, the developer has to create a configuration file containing the names of the nesC components or functions. As a first step, the developer links the run-time component `TinyTracerC` with the TinyOS application by wiring the `StdControl` interface of

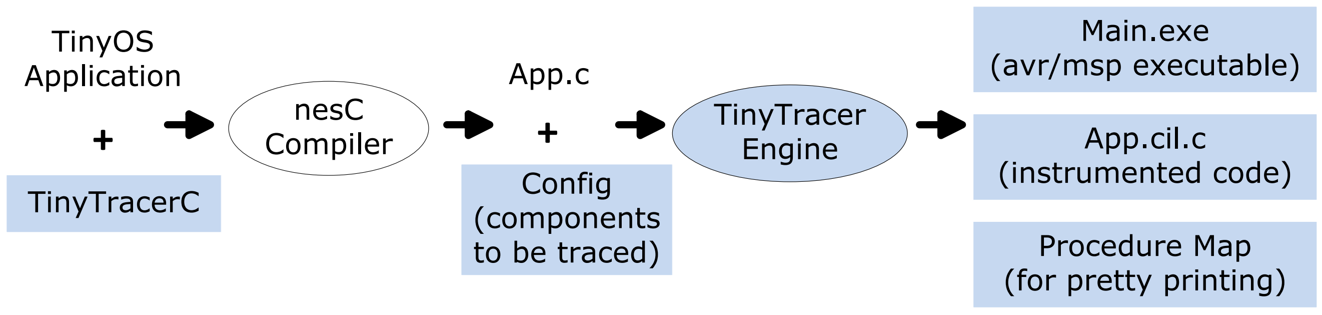

Figure 1: Workflow of TinyTracer at compile-time for TinyOS applications. The shaded components are parts of or outputs produced by TinyTracer

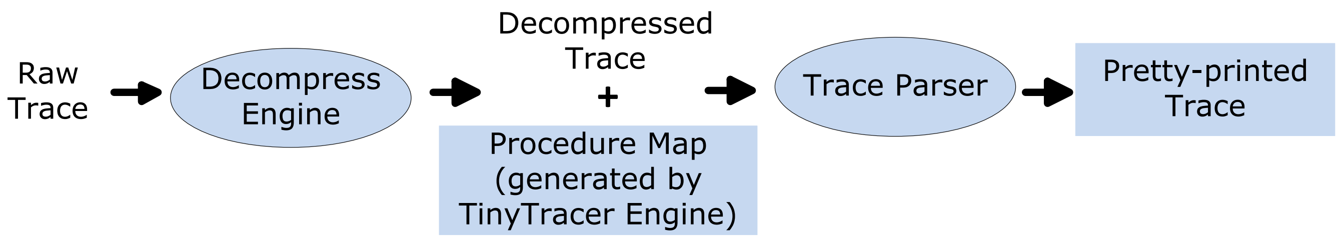

Figure 2: Workflow of TinyTracer at run-time. The shaded components are parts of or outputs produced by TinyTracer

application's main component to the TinyTracer run-time component. The nesC compiler produces a C file (`App.c`), which along with the configuration file is given as input to the TinyTracer engine. The TinyTracer engine produces a new C file (`App.cil.c`), which contains the instrumented code and a procedure map file, which is used for pretty printing. The TinyTracer engine can be configured to use the target platform compiler to compile `App.cil.c` and get the target platform specific executable.

`TinyTracerC` is implemented as a nesC component, which stores the trace generated at run-time in 2 in-memory trace buffers, each of length 192 bytes. It has a TinyOS task `CompressAndStoreTraceTask` that runs in the background and compresses the buffer, stores the compressed buffer into 16 flash pages each of length 16 bytes. These flash pages are drained by the `EEPROM` component in TinyOS. It is important to implement this task as a state machine that does some useful work and posts itself repeatedly, as we noticed long-running tasks can starve other executions including trace storage. Since the trace is constantly generated, `TinyTracerC` carefully coordinates the buffers usage with locks. The buffer sizes are configurable and depend on the application being traced.

The run-time workflow of TinyTracer is shown in Figure 2. The raw trace collected is decompressed using a decompressing engine. The decompressed trace is parsed by the trace parser, which uses the procedure map file to pretty-print the trace. An example of a pretty-printed trace for the `Oscilloscope` application in TinyOS is shown in Figure 3.

4. DEMO

During the demo, we will describe and demonstrate the compile-time as well as run-time workflow of TinyTracer to the users. The users will be able to choose the components of an application to trace and see the pretty printed trace.

```
<OscilloscopeM__Timer__fired> start
<OscilloscopeM__Timer__fired> end 0
<OscilloscopeM__ADC__dataReady> start
<OscilloscopeM__ADC__dataReady> end 3
   ... [the 4 lines above repeats 9 times]
<OscilloscopeM__dataTask> start
  <LedsC__Leds__yellowToggle> start
  <LedsC__Leds__yellowToggle> end 1
<OscilloscopeM__dataTask> end 0
<OscilloscopeM__DataMsg__sendDone> start
<OscilloscopeM__DataMsg__sendDone> end 0
```

Figure 3: Partial listing of pretty-printed trace

5. ACKNOWLEDGMENTS

This research is supported, in part, by the National Science Foundation (NSF) under grant 0834529. Any opinions, findings, conclusions, or recommendations in this paper are those of the authors and do not necessarily reflect the views of NSF.

6. REFERENCES

[1] J. Beutel, K. Römer, M. Ringwald, and M. Woehrle. Deployment techniques for wireless sensor networks. In G. Ferrari, editor, *Sensor Networks: Where Theory Meets Practice*, Heidelberg, 2009. Springer.

[2] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC*, 2002.

[3] V. Sundaram, P. Eugster, and X. Zhang. Efficient diagnostic tracing for wireless sensor networks. In *ACM SenSys*, 2010.