Zooming Into Radio Events by Bus Snooping

Zhitao He, Thiemo Voigt

Swedish Institute of Computer Science (SICS), Kista, Sweden

Abstract. In this position paper, we advocate the use of bus snooping to trace radio events. Highly precise and unintrusive, the technique leads to potentially more efficient code and enables more insightful protocol analysis than conventional code instrumentation techniques.

1 Introduction

Communication between Cooperating Objects is typically carried out over a multiple-layer protocol stack. The communication interface resides at the bottom of the stack, and its device driver implements packet transmission and reception routines as well as certain MAC primitives. Performance analysis of the device driver is typically done by insertion of instrumentation code, which logs API calls and interrupt events with a timestamping function provided by the host OS. For example, the Contiki OS's radio driver for the Tmote Sky platform can timestamp incoming packets at a default precision of 2.44 *ms*. Due to CPU loading concerns, the timestamps have a limited resolution; they also incur extra latency in the code execution path. Furthermore, tracing interactions between a pair of communicating motes requires accurate time synchronization, which entails a considerable increase in communication overhead. Contiki's *timesynch* protocol, for example, piggybacks a 3-byte timestamp construct to every data packet.

To alleviate both the precision constraint and the measurement overhead for communication performance analysis, we advocate an unintrusive bus snooping technique that performs event tracing on the communication interface. We attach a logic analyzer to the communication bus, which samples the pins' logic levels at a high rate. A timeline of command strobes, interrupt signals, data bits, and extra test signals can then be constructed over a test run, providing a rich amount of information to the developer for performance analysis or debugging purposes. The ability to simultaneously trace a set of signals, potentially selected from a pair of communicating motes, makes it particularly easy to detect events or event sequences triggered by protocol state transitions. One can zoom into a particular region of the timeline view, either for searching an event sequence or for gauging a code block's processing latency. Furthermore, the logged signal traces can be exported to a data file for advanced offline processing.

We show two use cases of our technique to highlight the productivity enhancements to a software developer. For all experiments, we use a USB logic analyzer [1] to snoop the pins of a pair of Tmote Sky motes running Contiki 2.4.

[1] Saleae Logic 16 logic analyzer. Web page: `http://www.saleae.com/logic16`

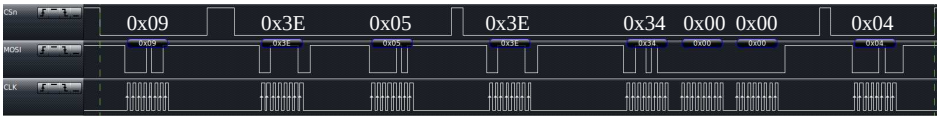
2 Optimizing Bus Latency

We analyze the bus events of a frequent elementary mote operation: packet transmission. A Contiki MAC protocol submits a packet to the radio driver for transmission, by calling a standard *radio_send(len, pkt)* method with a packet size parameter along with a payload pointer. The radio driver then constructs a PHY frame by adding headers and footers around the payload, and then transmits the frame. A merit of this generic *radio send* API is that implementation details of PHY frame construction and decoding are completely abstracted away by the device driver. Whether any performance penalty in terms of bus latency is entailed by this layer separation requires analysis of the specific device driver that implements the API. In general, bus latency can be divided into two components: a marginal cost per data unit, as a result of copying between the MCU's packet buffer and the radio's frame buffer; a fixed cost per packet due to signaling overhead. We analyze the performance of the CC2420 driver's *radio send* by snooping the control and data commands over the SPI bus during a call to the method. A trigger function of the logic analyzer takes us right to the beginning of event sequence, from where we can zoom in for precise timing analysis of the trace. Figure 1(a) shows a 80 μs trace section captured over a transmission of a 4-byte frame. The CSn signal is set low during each MCU access to CC2420; The MOSI signal is updated at the rising edges of the CLK signal, indicating serially transferred command/data bytes from the MCU to CC2420. The whole section can be broken down visually into four bus accesses, bounded by the three CSn spikes:

1. Issuing a command strobe to flush the TX FIFO: **0x09** in the figure.
2. Writing a one-byte PHY header to the TX FIFO: **0x3E** followed by **0x05** (PHY payload size).
3. Writing 3 payload bytes to the TX FIFO: **0x3E** followed by 3 data bytes from user pointer. (2-byte CRC checksum will be appended automatically by hardware before transmission.)
4. Issuing a command strobe to start frame transmission: **0x04**.

Step 2 and 3 issue the same **0x3E** command twice, one for writing the 1-byte PHY header and the other for writing the 3-byte payload, which results in a waste of bus bandwidth. Despite that the redundancy might as well be detected by careful code inspection through the CC2420 driver, our intuitive timeline view allows us to further measure the overhead to sub-μs precision, which would be unattainable with software instrumentation. By placing time markers at transition edges of the CSn signal, which signal the beginning and the end for each command access, we arrive at precise latencies of a FIFO write command: the fixed cost is 13.9 μs and the marginal cost per written byte is 6.6 μs. This means a 200% overhead for single byte accesses in this case.

To amortize the fixed cost for a FIFO write, we take full advantage of batch SPI transfers supported by CC2420 by combining step 2 and 3 into a single write command. This however obliges allocation of a single, contiguous packet buffer for storing both the PHY header and the payload ahead of the FIFO write. Such



(a) Default

(b) Fast

(c) Faster

Fig. 1. SPI bus latency for transferring a 4-byte frame. SPI clock rate = 2 MHz, sampling rate = 16 MHz

an optimization violates of the original PHY layer-independent API, but yields a considerable latency reduction, as shown in Figure 1(b). To achieve ultimate bus throughput, we further combine step 1's single-byte flush command strobe with step 2 write command into a continuous command sequence, by removing the chip deselect/select instructions between them, as shown in Figure 1(c). These two measures save us 20.5 μs per packet in total, which corresponds to 82 MCU cycles or 5 bit periods. We can make similar optimizations on the receiver path, reducing one-hop communication latency further and maximizing throughput.

3 Events Pattern Mining

Our event analysis technique can also be applied to study of random events, such as packet detections dependent on varying radio channel quality. Previous studies in reactive radio jamming have exploited CC2420's Start-of-Frame Delimiter (SFD) detection interrupt as a triggering signal for an eavesdropping jammer to transmit jamming signals [1] [2]. An important limitation of any reactive jammer though is a minimum switching time from listening mode to transmission mode, which sets a lower bound for the size of any jammable packet. A standard IEEE 802.15.4 acknowledgment frame consist of only 6 bytes, which is too short to be jammable by existing reactive jammers based on SFD decoding.

The problem can be alleviated by reducing the switching time, which depends on whether we can find a new triggering signal that becomes available earlier than the SFD interrupt, i.e., some sort of preamble energy indicator. During a search of such an indicator, we focus our attention to a signal output by the radio receiver's automatic gain control (AGC) circuit. We conjecture that the frequency that the AGC circuit updates its gain correlates somehow to changes

in the received signal strength. We configure the CCA pin of CC2420 to output the internal signal *AGC_UPDATE*, which manifests as a high one 16 MHz clock cycle each time the AGC gain is updated. The 4 MHz MCU on Tmote Sky is too slow to capture these narrow 16 MHz spikes. We instead tap a probe from our logic analyzer to the pin, thus are able to capture occurrences of this random signal with a precision of 0.01 μs.

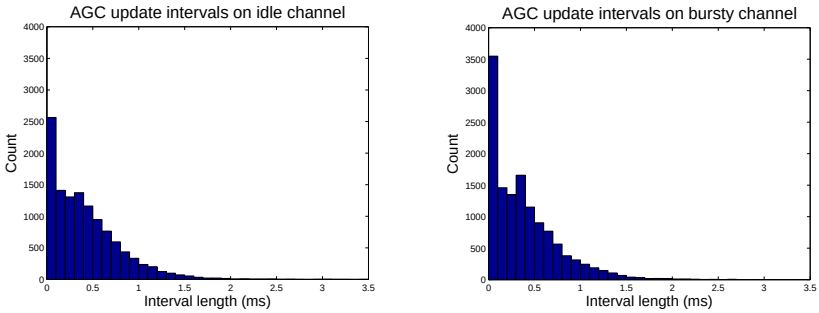
Our Tmote Sky listening to an idle channel observes 11822 AGC updates in just 5 seconds. Time intervals between each two consecutive updates range between [4.62 μs, 3.45 *ms*], with a mean of 0.425 *ms*. The statistical distribution of the update intervals are shown as a histogram in Figure 2(a).

Despite the high frequency of AGC updates and apparently random intervals between them, we want to further investigate whether a packet triggers any extra updates. We configured another Tmote Sky to send a burst of 320 packets, at 64 pkts/s, while repeating the previous measurement on the listening mote. The histogram in Figure 2(b) shows an increase of short update intervals. If one zooms into the trace to examine pin activities preceding each SFD interrupt, an interesting pattern of the AGC updates can be observed: approximately 120 μs to 160 μs before frame detection, a burst of two or more updates occur in short intervals ranged from 4.62 μs to 20 μs. Since this burst pattern occurs during the known period of the 4-byte frame preamble, it might qualify as a preamble indicator useful for frame prediction.

We set out to design and implement a high-pass filter in the time domain that extracts bursty AGC updates from the pool of sporadic updates. First, we export AGC and SFD pin transition events, all timestamped at 0.01 μs precision, from the logic analyzer program to a CSV file. We then load the file into a MATLAB script that filters the AGC events and attempts to match the resultant AGC update bursts to corresponding SFD events. A match indicates successful frame prediction, whereas a mismatch indicates a false prediction. To emulate 6 different signal-to-noise ratios at the receiver, we step down the transmission power over successive test runs. We initially use a narrow interval filter of just 5 μs, then repeat the tests using a 10 μs filter. Figure 3 shows that the prediction rate is close to 100% at presence of a strong signal, but drops as the signal weakens, while false predictions increase. Comparison between Figure 3(a) and Figure 3(b) shows that the 10 μs filter yields a higher prediction rate for weak frames, albeit with higher likelihood of false predictions.

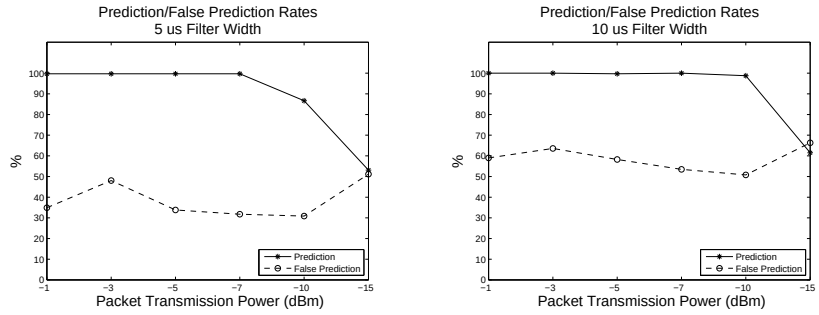
4 Limitations

There are a number of limitations imposed by the use of a logic analyzer. The number of available channels caps the number of concurrent test signals. Our 16-channel logic analyzer thus can monitor at most four 4-wire SPI buses at the same time. The bandwidth capacity of the logic analyzer limits the sampling rate of each channel, which will become an an issue if buses of higher data rates are to be snooped. The trace length is limited by the user's free disk space.



(a) Idle Channel (b) Bursty Channel

Fig. 2. Histogram of AGC updates intervals



(a) (b)

Fig. 3. The AGC update bursts can be used as a frame predictor with a very high prediction rate, given a strong received signal.

5 Conclusions

In this paper, we show how bus snooping can gain us new insights into low level radio events. This bottom-up approach is precise and unintrusive, compared with conventional software instrumentation. We see potential opportunities to extend the use of this technique for debugging and performance analysis at higher layers.

Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 224282 and has been partially supported by the FP7 NoE CONET.

References

1. Wood, A., Stankovic, J., Zhou, G.: DEEJAM: Defeating Energy Efficient Jamming in IEEE 802.15.4-based Wireless Networks. Secon (2007)
2. He, Z., Voigt, T.: Precise Packet Loss Pattern Generation by Intentional Interference PWSN (2011)