AVEKSHA: A Hardware-Software Approach for Non-intrusive Tracing and Profiling of Wireless Embedded Systems

Abstract

It is important to get an idea of the events occurring in an embedded wireless node when it is deployed in the field, away from the convenience of an interactive debugger. Such visibility can be useful for post-deployment testing, replay-based debugging, and for performance and energy profiling of various software components. Prior software-based solutions to address this problem have incurred high execution overhead and intrusiveness. The intrusiveness changes the intrinsic timing behavior of the application, thereby reducing the fidelity of the collected profile. Prior hardware-based solutions have involved the use of dedicated ASICs or other tightly coupled changes to the embedded node's processor, which significantly limits their applicability.

In this paper, we present AVEKSHA, a hardware-software approach for achieving the above goals in a non-intrusive manner. Our approach is based on the key insight that most embedded processors have an on-chip debug module (which has traditionally been used for interactive debugging) that provides significant visibility into the internal state of the processor. We design a debug board that interfaces with the on-chip debug module of an embedded node's processor through the JTAG port and provides three modes of event logging and tracing: breakpoint, watchpoint, and program counter polling. Using expressive triggers that the on-chip debug module supports, AVEKSHA can watch for, and record, a variety of programmable events of interest. A key feature of AVEKSHA is that the target processor does not have to be stopped during event logging (in the last two of the three modes), subject to a limit on the rate at which logged events occur. AVEKSHA also performs power monitoring of the embedded wireless node and, importantly, enables power consumption data to be correlated to events of interest.

AVEKSHA is an operating system-agnostic solution. We demonstrate its functionality and performance using three applications running on the TelosB motes; two in TinyOS and one in Contiki. We show that AVEKSHA can trace tasks and other generic events and can perform energy monitoring at function and task-level granularity. We also describe how we used AVEKSHA to find a subtle bug in the TinyOS low power listening protocol.

1 Introduction

It is often important to get an idea of the events occurring in an embedded wireless node when it is deployed in the field in a remote location, away from the convenience of an interactive debugger. Such visibility can be useful for various purposes — for debugging any problem *a posteriori* in the lab, by recreating the exact sequence of events that the node experienced in the deployment (this approach is called "record and replay-based debugging") [1, 2]; for profiling the operation of a node for the performance of its various software components and the energy consumed by different hardware and software components on the node [3, 4, 5]. As an example of the latter use case, a system owner may be interested in figuring out which software component is being invoked most often and which software component is consuming most energy per invocation. It is often not possible to do these determinations in a lab setting because the events that the node experiences in the deployment cannot be recreated in the lab and the events (and even their sequence) can have a bearing on these questions.

We would like to have visibility at a fine granularity - both spatially and temporally. Spatially fine visibility implies that it should be possible to trace *individual* events of interest as opposed to only bursts of events (clearly, tracing *every* event is likely to be prohibitive) and it should be possible to trace performance and energy at fine code regions, such as a function or a task (using TinyOS terminology). This is desirable because the fine region of code can then be debugged if it is determined through performance profiling that this region is causing a performance bottleneck, through energy profiling that it is consuming unexpectedly large amounts of energy, or through record and replay that it is the source of a bug. Temporally fine visibility implies that it should be possible to do the tracing with a high sampling frequency. Clearly, the two dimensions are not independent. In order to trace small regions of code in a loop, it is necessary to be able to trace at a fine temporal granularity.

While the problem motivation laid out above has been clear to researchers for quite some time [6], it has proved very difficult to provide a solution for low-cost embedded wireless nodes that can operate at a large deployed scale. The first line of attack has been to provide pure software solutions [6, 7, 8, 1, 2]. Such solutions have perturbed the application too much to be useful for many of the use cases indicated above. For one, they change the timing behavior enough that some bugs get suppressed. Else, they cause such a large slowdown in the application execution that it is not possible to employ them in a deployed setting. To get around this problem, a recent software solution [2] has focused on a specific kind of tracing (control flow tracing) and intelligent static analysis and runtime trace collection, compression and storage. Thus, it addresses one of the above usage scenarios. The second line of research has developed hardware solutions for subsets of the usage scenarios laid out above. For example, [5] has developed a dedicated integrated circuit, implemented using an FPGA, that is tightly integrated with the host processor and its peripherals and can measure energy drawn accurately at millisecond resolution. Quanto [4] is a solution that de-emphasizes sophisticated hardware design. Instead, it measures energy at the node level, uses indication from device drivers about changes in power state, and performs causality tracking to pin down energy usage due to individual activities. Thus, Quanto is a hardware-software solution, and like all prior solutions that have a software part, is OS-specific (in this case, TinyOS).

A high-end hardware solution for tracing the execution on an embedded processor is provided by solutions such as Green Hills Software's SuperTrace probe and TimeMachine tools [9]. These solutions can collect fine-grained trace data from nearly all 32-bit and 64-bit processors, even those without integrated trace hardware. Unfortunately, such solutions are very expensive in dollar terms (*e.g.,* the SuperTrace probe and TimeMachine tools together cost almost $15,000) and are not available for the low-end embedded processors that are commonly used in embedded wireless nodes.

In summary, our problem statement is the following: *How to perform non-intrusive tracing of execution at a high spatial and temporal granularity suitable for an embedded wireless node, i.e., in a low-cost manner and one that can be deployed at a large scale?*

In this paper, we present AVEKSHA, a system that achieves this goal[1]. AVEKSHA is based on an insight that most processors, including low-cost embedded processors, offer visibility into their internal workings through an On-Chip Debug Module (OCDM), whose signals are exposed through a standard JTAG interface. This interface has been used by embedded system engineers primarily for interactive debugging, such as single stepping, showing values of registers, *etc*. We show how this visibility, together with the fact that most OCDMs provide a general-purpose method of setting triggers, can be leveraged in AVEKSHA to perform automated tracing in a deployed setting.

We develop a debug board formed of standardized components – a microcontroller unit (MCU), which in our case happens to be the same as the application processor, MSP430F1611 from Texas Instruments, and an Actel FPGA, both of which interact with the OCDM on the application processor over the JTAG interface. We refer to our debug board as the Telos Debug Board (TDB) because it is intended to be used with the Telos wireless sensor node (however, that out solution is not restricted to the Telos and can easily be adapted to other embedded platforms based on the MSP430 microcontroller, and with some effort to other embedded platforms). The MSP430 OCDM (also referred to by the microcontroller datasheets as the Enhanced Emulation Module or EEM) allows AVEKSHA unprecedented visibility into the state of the application processor. Further, the OCDM has a small circular buffer where events of interest can be stored and subsequently drained by the FPGA on the TDB. The triggering mechanism of the OCDM is very flexible and is therefore attractive for AVEKSHA. For example, the OCDM can be triggered to indicate when the application processor has accessed a certain memory region or a certain peripheral device, such as a sensor. We find that the triggering mechanism can be combined with thoughtful design to trace all the events of interest for our three usage scenarios – performance profiling, energy profiling, and record-and-replay.

One challenge that we face, and resolve partially, is the need to do real-time tracing, *i.e.,* without interrupting the application processor. AVEKSHA is able to achieve this when the rate of events that it has to trace does not exceed some bound, which depends on the mode of tracing it uses. AVEKSHA operates in one of three modes: *breakpoint*, *watchpoint*, and *program counter (PC) polling*. Breakpoint is a baseline and we use it for demonstrating some functionality of the TDB. It is intrusive and, therefore, does not meet our solution requirements. The *watchpoint* mode has AVEKSHA set triggers, where each trigger unambiguously maps to an event of interest (such as when a sensor is read). When a trigger fires, the application processor is not stopped, but the state is dumped to a buffer on the OCDM, which is then emptied out by AVEKSHA. This is a rate-limited operation and if events of interest happen with a high enough frequency, the buffer overflows and AVEKSHA misses some events of interest. In the *PC polling* mode, the TDB tracks the program counter values of the application processor without interrupting it. Then, it processes the PC values to determine events of interest, such as when control flow has entered a particular function. These three modes reveal different tradeoffs in terms of intrusiveness, the flexibility in defining which events to collect, and the rate at which collection can be done.

We make the following claims to novelty and practical feasibility from AVEKSHA:

1. We present the first technique for non-intrusive tracing of a wide variety of events, including arbitrary user-defined events, in embedded wireless nodes. We motivate the events of interest from three well-accepted usage scenarios.
2. Our tracing technique is agnostic to the operating system, compiler infrastructure, or language in which the application is implemented.
3. Our hardware is built using off-the-shelf components and requires little effort in designing and integrating. The hardware of the application board is modified only very slightly for enabling the tracing.
4. Our solution is suitable for deployment at a large scale because it is low cost, can operate on battery power, and extracts program information from the application processor without needing to halt it.

Our solution also has some limitations. We provide a detailed discussion of these, along with thoughts on how to mitigate them, in Section 5. In brief, the TDB is a relative energy hog itself, drawing about the same power as the application processor board. Its ability to keep pace with events is exceeded if a burst of 8 events happens within a window smaller than 976 clock cycles (in the watchpoint mode) or events happen more frequently than 7 clock cycles (in the PC polling mode).

[1] AVEKSHA is a Sanskrit word that means "to monitor".

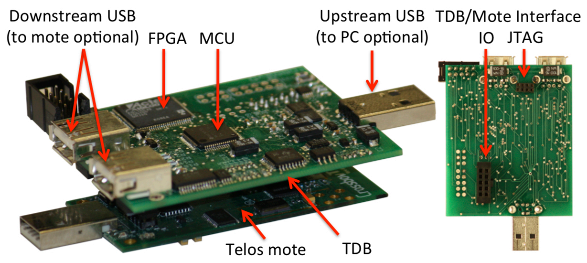

Figure 1. The Telos Debug Board (TDB) shown with a Telos mote underneath. The underside of the TDB is also shown on the right.

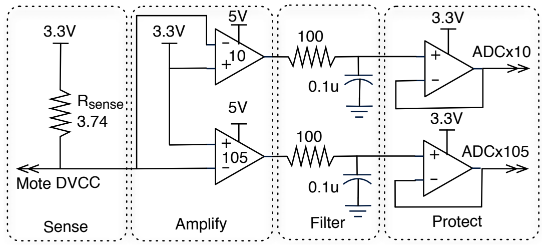

Figure 2. A simplified schematic of the energy monitoring circuit of the TDB.

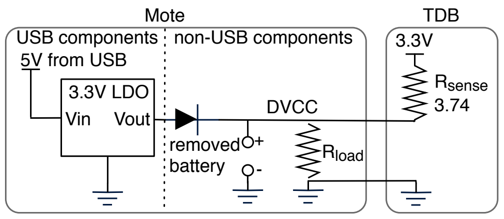

Figure 3. A simplified schematic of powering the mote through the TDB. The battery should not be connected to the mote.

The roadmap for the rest of the paper is as follows. In Section 2, we describe the hardware of the TDB and the firmware that goes on it. In Section 3, we show how the TDB can be used for profiling. In Section 4, we present the experimental setup and results with two TinyOS and one Contiki application. In Section 5, we discuss feasible extensions of AVEKSHA. In Section 6, we review related work and Section 7 concludes the paper.

2 TDB Hardware and Firmware

In this section, we present the design of the Telos Debug Board (TDB), which operates as a sister board to the Telos mote, and provides JTAG control, and through it, energy monitoring and execution profiling. An MCU and an FPGA provide the programmability of the TDB.

Terminology. We lay out some terminology that we will use through the rest of the paper. We wish to monitor the execution of the *application processor* that is part of an *application processor board*. The application processor board, which we sometimes also refer to as the *mote*, has various peripherals such as sensors and the JTAG interface in addition to the application processor. We refer to the hardware board that is a part of our solution as the Telos Debug Board, while the entire hardware-firmware that forms our solution is called AVEKSHA.

Figure 1 shows a photograph of the TDB with a Telos mote attached underneath. The MCU, FPGA, and multiple USB ports on the TDB are highlighted. The figure also shows the JTAG and IO connections between the TDB and the mote. The TDB is designed so that it can be deployed in the field, connected to a mote. In this mode of operation, a battery powers the TDB, which in turn provides power to the mote. As a secondary mode of operation, the TDB can also stream logged events directly to a USB host such as a laptop. This is useful for in-lab debugging.

2.1 Energy Monitoring

Energy is a key concern for sensor networks, because motes must operate unattended on battery power for long periods of time. When optimizing an application to reduce energy consumption, it can be useful to observe how much energy is consumed in different states. The TDB can measure and log the power consumed by the mote, which can then be correlated to different operational states of the mote.

The standard method for measuring energy consumption is by monitoring the voltage of a sense resistor. The sense resistor (R_{sense}) is placed in series with the load being measured. The voltage (V) across R_{sense} is sampled and the load's current draw is calculated by $I = V/R_{sense}$. The supply voltage (V_{supply}) can then be used to find the power being drawn by the load with $P = I * V_{supply}$. The power samples can then be integrated over time to determine energy consumption.

The challenges presented in monitoring energy in sensor networks is the wide dynamic range of power draw of a mote and the rapid changes in power draw. For example, a mote may draw only tens of μA in sleep mode and as much as $30mA$ when fully active. It is may not be sufficient to ignore the small power draw when the mote is in low power mode, because typical sensor network applications spend long periods of time in the low power state while only waking for short periods of time. Further, the change in the current draw when the mote transitions from one state to another is rapid.

To meet these challenges, we use two instrumentation amplifiers to amplify the voltage across R_{sense} by a gain of 10 and 105, as shown in Figure 2. Because R_{sense} is placed on the high side, the amplifiers need to be supplied with a voltage larger than 3.3V, in this case 5V. The output of these amplifiers is fed into an RC low-pass filter with a cut off frequency of about 16KHz. Another pair of amplifiers with unity gain is used to protect the ADCs which cannot tolerate more than 3.3V. Two ADC channels of the MCU on the TDB sample the x10 and x105 lines at 20KHz. The ADCs are 12-bit and sample against a reference voltage of 2.5V. This gives ADCx10 a resolution of 61μV across R_{sense} which is equivalent to 16.3μA of current draw, and a maximum reading of 250mV or 66.8mA, which is more than the mote's maximum current draw of 30mA. The ADCx105 gives a resolution of 5.81μV across R_{sense} which is equivalent to 1.55μA of current draw, and a maximum reading of 23.8mV or 6.366mA.

As shown in Figure 3, R_{sense} is placed between the TDB's 3.3V supply and the mote's DVCC line, while the ground

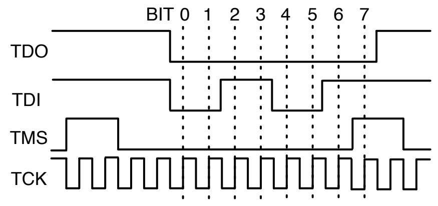

Figure 4. Timing example for shift IR. The byte 0xCC is shifted in on TDI while 0x00 is shifted out on TDO, least significant bit first.

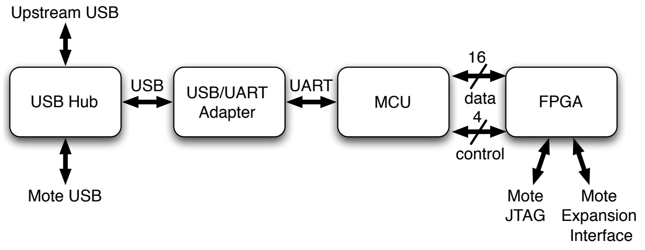

Figure 5. Hardware architecture of the TDB.

lines of the mote and the TDB are shared. This configuration is known as high side sensing. One advantage of high side sensing is that the ground plane of the mote and the TDB are shared. If on the other hand, we had used a low side sensing approach, the ground of the mote would have been connected to one end of the resistor away from the ground. This would have caused the mote's ground not to be exactly at 0 V. Another advantage is that the Zener diode achieves isolation between the USB components and the non-USB components of the mote and thus allows us to capture the current draw of the non-USB components (which is what we are interested in) even when the mote is plugged into a USB port of a laptop (or some other computing platform). This is because the diode has a forward bias of about 360mV, meaning that as long as the voltage drop across the sense resistor remains below 360mV all current drawn by the non-USB part of the mote will be from the TDB and not the USB part of the mote. R_{sense} is chosen sufficiently low such that this will happen even at maximum power draw by the mote. The maximum current of the mote is 30mA which would result in a voltage drop across the sense resistor of 112mV.

To account for amplifier offset we use a switch between R_{sense} and DVCC that allows 3.3V to be temporarily placed at both ends of R_{sense} in a manner similar to [3]. The MCU has an ADC buffer of 16 samples. The x10 and x105 amplified signals are sampled alternately at a rate of 40ksps, to achieve an effective sampling rate of 20ksps. When the ADC buffer is full, an interrupt service routine sums up the samples in the buffer. A count of ADCx10, ADCx105, and the total number of samples is maintained. From this, the total energy consumed by the mote can be computed. It is desirable to use the ADCx105 reading due to its greater current resolution, unless there has been an overflow in its reading. This is determined by first reading the ADCx10 value and taking the ADCx10 reading only if it indicates an overflow in the ADCx105.

2.2 JTAG: Background

The application processor contains an on-chip debug module. This module can be used to emulate the processor (directly control the processor operations) and it can execute breakpoints and watchpoints when certain conditions of the data and address buses are met. A breakpoint halts execution of the processor, while a watchpoint records the contents of the data and address bus into an 8 entry circular buffer. The OCDM is implemented as a state machine that is controlled via the standard JTAG protocol.

JTAG uses four lines: data output to host (TDO), data input to target (TDI), mode select (TMS), and clock (TCK). JTAG shifts frames of data into and out of the OCDM and that changes the state of the OCDM. There are two basic shift modes: an 8-bit instruction register (IR) shift and an *n*-bit data register (DR) shift. The TMS line selects between IR or DR at the start of a shift based on the number of TCK rising edges for which it remains high. For example, in Figure 4, TMS remains high for two rising edges which selects the IR mode, while one rising edge would select the DR mode. The number of bits shifted in an n-bit DR shift is determined by TMS being high a second time during the shift of the last bit. Bits are shifted from the mote to the TDB on the TDO line and from the TDB to the mote on the TDI line. Although the JTAG protocol is standard, the sequence of instructions that must be shifted into the OCDM on the MSP430 is proprietary. We have reverse engineered these control sequences and used them in AVEKSHA to determine what command sequences must be sent to the OCDM for the application to enter a breakpoint, to set a watchpoint, or to enable PC polling.

2.3 Architecture: Hardware

As shown in Figure 5, the TDB consists of a USB hub, a USB to UART adapter, an MCU, and an FPGA. The USB components are primarily for use in a lab environment and provide reprogramming, control, and streaming of log data. They can remain unpowered when the debug board is deployed in the field. The USB hub has 1 upstream port and 3 downstream ports. The upstream port is used to access the debug board from a PC. One of the downstream ports is permanently connected to a USB to UART adapter that provides reprogramming and data transfer to and from the MCU. The second downstream port is available for connection to the USB port on the mote. This is useful in a lab or testbed deployment where access to the mote's USB port is desired. The final downstream port is available for future use, and we envision it being used in testbed deployments to daisy chain several TDBs together.

The FPGA interfaces with the mote's JTAG and expansion interfaces. The expansion interface of the mote provides access to the some of the its UART, I2C, ADC, and GPIO peripherals. It was necessary to use an FPGA to control the mote's JTAG to be able to poll at a sufficient frequency to keep up with the events we want to observe. For some operations (such as PC polling) we have found the need to drive the clock line of the JTAG up to 24MHz. To implement a protocol in software that can drive the JTAG at 24MHz would require a processor that operates at several times that speed.

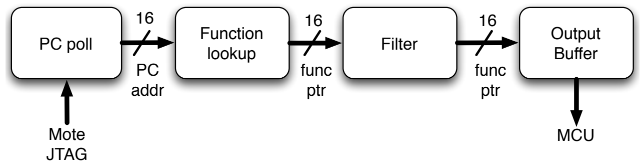

Figure 6. FPGA pipeline in PC polling mode.

Additionally, there is the problem of processing the collected JTAG data to determine what should be logged. The FPGA allows pipelining of JTAG control and data processing, so that the polling loop never waits for data processing.

An MCU is placed between the USB to UART adapter and the FPGA. It performs tasks that are less time critical and better suited to software. For example, initialization of the mote for debugging, reading the contents of the mote's program memory, and disassembly of the mote's program code are performed by the MCU. Using an MCU also makes adding functionality to the debug board easier because the MCU can be reprogrammed over USB. To simplify programming, the MCU used on the TDB is an MSP430 processor that is identical to the one used on the Telos mote. The TDB operates its MSP430 MCU at 8MHz.

In the current prototype, the TDB logs are streamed over USB. It would also be possible to add some Flash memory to act as a circular buffer as was done in FlashBox [8]. Our maximum reliable streaming throughput over USB is 1Mbps. This is limited by the USB 1.0 hub and adapter, which have a theoretical throughput of 1.5Mbps. Moving to USB 2.0 would boost the USB throughput to 480Mbps, which would make the 8MHz MCU the bottleneck in streaming logs. However, we have found 1Mbps to be sufficient in all of our experiments as long as buffers are added in the MCU and FPGA to absorb short bursts of data to be logged.

2.4 Architecture: Firmware

The firmware of the TDB consists of C code for the MCU and Verilog code for the FPGA.

Firmware on the MCU The MCU is responsible for initialization tasks. When the TDB is first connected to the mote, the MCU sets the FPGA into a mode where the MCU can directly control the JTAG lines connected to the mote. Through JTAG commands, the mote is put into a halt state and the program memory of the mote is read. A simple disassembly of the program is performed, where the start of every function block is discovered by examining the destination of every call instruction in the code. The resulting table is then programmed into the FPGA's RAM for use by the function lookup module. At this point, any watchpoint trigger can be set on the mote. What triggers will be set will depend on the goal of the tracing. For example, if record and replay is desired, then triggers will have to be set for the entry point of every function, entry point of every interrupt handler, and read from any peripheral device. We explain in Section 3 the complete list of triggers that can be supported. Finally, the MCU sets the FPGA to either the PC polling or the watchpoint mode and resumes execution on the mote. Thus, the MCU on the TDB functions as an orchestrator, but leaves the core functionality for the FPGA.

The main advantage of reading and disassembling the mote's program memory when the TDB is connected to the mote is that the TDB need not be aware *a priori* of what application the mote is running. The TDB can perform this process independent of the application or operating system used by the mote, but based on the machine language of the MSP430.In addition to finding the entry point of every function, the disassembly of the code is also used to record the start of every interrupt service routine, the address of every function call, the address of every function return, the address of return from every interrupt service routine, the contents of the interrupt vector table, and the addresses of special `nop` instructions (e.g. `MOV R4, R4`), that are used as trigger markers in the code.

Firmware on the FPGA The FPGA is responsible for talking to the OCDM on the mote's MCU through the JTAG port. The FPGA polls the OCDM to to detect the occurrence of any triggers of interest. Following each iteration of the polling loop, processing may need to be performed to decide whether or not the polled data should be logged. For example, with PC polling, a log entry should be generated when the polled PC value falls into the address range of a new function. To prevent the slowdown of polling, the processing is pipelined in the FPGA. Figure 6 shows the pipeline for PC polling. The FPGA due to its inherent parallelism can support this pipeline. At the end of each PC poll, the PC address is passed to the function lookup module. The function lookup module contains a table in RAM of the start address of every function block. A binary search is performed on the table to find the start address of the function block that corresponds to the polled PC address. The function table capacity is 1024 function pointers, so the lookup completes in at most $log_2(1024) = 10$ reads from RAM, which completes well before a single PC poll. Note that one cycle on the FPGA is much faster than one cycle on the application processor (20 MHz versus 4 MHz for our experiments) and hence, the address lookup does not become a bottleneck under experimental conditions. After the correct function pointer is discovered, it is passed to a filtering module. This module decides whether or not the function pointer should be logged. If the function pointer corresponds to the same function as the last logged one, then it does not need to be logged again. A new function pointer indicates that a different function has been entered, so the value is logged. Finally, function pointers that are to be logged are passed to a FIFO output buffer maintained in the FPGA. The MCU on the TDB reads this buffer and logs the data either in local Flash memory, or, if required, streams the log entries to a host machine to which the TDB is connected through a USB port or through wireless communication. The buffer is necessary because the MCU performs other functions, such as energy monitoring, and may not be able to read a value to be logged in the time it takes to perform a single PC poll. The buffer also absorbs peaks in the rate of new functions being invoked. We have observed a buffer size of 256 to be sufficient to absorb all peaks in the programs that we have monitored.

For some pathological cases, it is possible that the PC polling mode of AVEKSHA is thrown off—the execution may transition from function `func1` to `func2` and then back to

Table 1. Types of triggers available for monitoring events.

Event	Condition	# Triggers
Function call	MDB-F==0x12B0	1
Function return	MDB-F==0x4130	1
Interrupt	MAB-R≥0xFFE0	1
Interrupt return	MDB-F==0x1300	1
Peripheral read	0x0010≤MAB-R≤0x01FF	2
Peripheral write	0x0010≤MAB-W≤0x01FF	2
User defined	MDB-F==0x4404	1

`func1`. The function `func2` is small enough that PC polling misses the transition and mistakenly determines that the execution has stayed in `func1` all through. For this to happen, `func2` has to be smaller than 7 clock cycles based on the timings of PC polling, which we detail in Section 4.1.

3 Using AVEKSHA for Tracing and Profiling

There are three modes that AVEKSHA can operate in while monitoring application execution, namely *Breakpoint mode*, *Watchpoint mode*, and *PC Polling mode*. Depending on the mode of operation, AVEKSHA interacts with the OCDM on the application processor in different ways. Therefore, these modes have different tradeoffs in terms of the level of intrusiveness to the application (breakpoints are the most intrusive), the flexibility offered in terms of the kinds of events that can be observed (watchpoints are the most flexible), and the speed of event logging (PC polling is the fastest). Before we describe the three modes of operation, we discuss the kinds of triggers that AVEKSHA can set for observing events of interest on the application processor.

3.1 Types of Triggers Available

The OCDM on the application processor allows us to set 8 concurrent triggers for detecting events of interest. Although this number may, upon first glance, seem insufficient to create a complete profile of an application, that is not the case because the MSP430 offers far more advanced triggers than just the program counter (PC) reaching a particular value. For example, a trigger can compare the Memory Data Bus (MDB) or the Memory Address Bus (MAB) to a set value or range of values. Additionally, the trigger can be restricted to be active only during an instruction fetch (F), a memory read instruction (R), or a memory write instruction (W). This gives us great flexibility in using these 8 concurrent triggers to capture all our events of interest.

All of the triggers that we use in this paper are listed in Table 1. The notation used for specifying the condition that the value on the Memory Data Bus equals `0x12B0` on an instruction fetch is given by: `MDB-F==0x12B0`. This particular trigger will fire for every function call because `0x12B0` is the machine code for a function call instruction. Similarly, the machine code for the return instruction from a function call is `ret=0x4130`. Therefore, the trigger `MDB-F==0x4130` will trigger on all function call return events. A call to an interrupt can be detected with the trigger `MAB-R`≥`0xFFE0`. The interrupt vector table is located between address `0xFFE0` and the end of the address space at `0xFFFF`. Every time an interrupt is to be serviced, the processor reads the interrupt vector table to determine the address of the interrupt service routine that corresponds to the interrupt being serviced. Interrupts have their own return instruction (`reti=0x1300`) that can be monitored with the trigger `MDB-F==0x1300`.

The compound trigger `0x0010`≤`MAB-R`≤`0x01FF` will fire for every read to memory between addresses `0x0010` and `0x01FF`. A compound trigger, such as the above, that contains two conditions is made by joining two triggers together, and uses 2 of the 8 available trigger entries. In the MSP430, the peripherals are all memory mapped to addresses between `0x0010` and `0x01FF`. The peripherals include any sensors that may be attached to the applicaton processor. For purposes of deterministic record and replay, it is important to track what sensor values are read. This can be done by using the trigger `0x0010`≤`MAB-R`≤`0x01FF`, which captures a read from the memory-mapped peripheral portion of memory. When a trigger is fired, the OCDM stores the values of the MAB and the MDB to the 8-entry circular buffer. The stored MDB will contain the value that was read from the peripheral.

While functions and interrupts are interesting points for monitoring, we would like even more flexibility to monitor any arbitrary event in the executing application. For example, if we want to monitor the execution of every `task` in TinyOS, we cannot do this with a function call trigger. This is because the `gcc` compiler inlines many of the tasks in the scheduler's `runTask()` function. One solution is to set the `noinline` directive on all task functions. We have verified that this works, however, this is unsatisfactory because it sacrifices the efficiency gains obtained due to function call inlining. A less costly solution is to trigger on a `nop` instruction. However, the MSP430 does not have an explicit `nop` instruction. Instead, compilers emulate this instruction by using a 1 cycle instruction that has no direct effect and no side effect on status or mode bits – specifically, `gcc` uses the instruction `MOV R3, R3` to emulate a `nop`. There are three possible 1 cycle instructions that meet the requirements for no effect or side effect: (`MOV Rn, Rn`), (`BIC #0, Rn`), and (`BIS #0, Rn`). With 16 registers available on the MSP430, this gives us 48 possible choices for an emulated `nop`. We can use different application-specific meanings for each emulated `nop` instruction to monitor 48 arbitrary events of interest. For our purposes we choose just one, (`MOV R4, R4`), which translates to the machine code `0x4404`, and add an instruction fetch trigger `MDB-F==0x4404`. A programmer can now place the assembly code (`MOV R4, R4`) at arbitrary places in the code to monitor user-defined events of interest, such as the beginning of a task.

3.2 Breakpoint Mode

Any of the 8 concurrent triggers available can be set as a breakpoint. When a breakpoint is reached, the OCDM on the application processor halts execution and takes control of the application processor. The TDB performs a continuous poll of the CPU state of the application processor. When it sees that the CPU state is halted, it retrieves the state of the application processor (*e.g.,* the value of the PC) and sends a command to the OCDM asking it to resume execution.

Table 2 shows the speed at which a single poll (or test) of the CPU state can be performed, the time it takes to read the PC register, and the time it takes to resume CPU execution.

All of these operations involve shifting values into the instruction register (IR) and data register (DR) of the OCDM. For example, a test of the CPU state requires shifting one IR and one DR, reading the PC register requires shifting 2 IRs and 4 DRs, and resuming the CPU involves shifting 3 IRs and 1 DR. The operations needed for achieving the tasks are not documented and we determined them through reverse engineering TI's IAR debug interface [10]. An IR can be shifted in 15 cycles of the JTAG clock (TCK) and a DR can be shifted in 23 JTAG clock cycles. Table 2 shows the times for performing the required shift operations in software with the MCU on the TDB running at 8MHz, and the FPGA implementation. The FPGA is able to operate the JTAG clock (TCK) at 10MHz, which is the fastest we have been able to operate the JTAG reliably for the breakpoint and the watchpoint modes and is the maximum rated speed according to the JTAG specification.

Breakpoints have the advantage that we never miss a trigger firing, because every time a trigger is reached the CPU is halted and control is passed to the TDB. The disadvantage of the breakpoint mode is that we lose the property of non-intrusiveness. In TinyOS, the MSP430 on the TelosB is set by default to operate at 4MHz meaning $1\mu s = 4 cycles$. Using the FPGA implementation, the time to poll and resume the application processor is equivalent to 91.2 cycles of the application processor. The application processor has to be halted for atleast this time while processing an event.

3.3 Watchpoint Mode

The same kinds of triggers as in the breakpoint mode can also be set as watchpoints. The JTAG interface has an 8-entry circular buffer where memory address bus (MAB) and memory data bus (MDB) are stored when a watchpoint is hit. As indicated earlier in Section 3.1, the trigger can be on an instruction fetch, read, or write. Thus, by recording the address bus content on an instruction fetch, it is possible to know the PC value. The most recent entry written to the 8-entry buffer is indicated with a set flag. The TDB polls the flag of the most recently written entry until it is cleared, indicating that a new entry has been written to the buffer. The TDB then continues to read entries until it again reaches one that has its last entry flag set.

Watchpoints have the benefit that unlike breakpoints, they are not intrusive to the application. The application does not have to interrupt its execution when a watchpoint trigger is met. However, this also means that there is a threshold for the rate of triggers that the TDB can keep up with. Beyond this rate, the circular buffer will wrap around and the TDB will miss some events of interest. Based on our empirical measurements (given in Table 2), the entire processing for one invocation of of a watchpoint trigger takes $30.5\mu s$, which corresponds to 122 cycles for the application processor (at the default frequency of 4 MHz). Thus, as long as we do not have a sustained burst of 8 events of interest within $8 \times 122 = 976$ cycles, the TDB in the watchpoint mode will not miss any event.

3.4 PC Polling Mode

The final approach to trace generation, is to forego using triggers entirely, and instead poll the program counter of the application processor as it is running. If we could observe every time the PC is modified, we could produce a perfect trace of program execution. This is practically infeasible, however, because it is not possible to do PC polling at a high enough rate and the amount of trace data generated will be too large to be stored or streamed back to a base station. However, for our applications, we find that it is possible to do PC polling at a fast enough rate to keep track of all the the functions that execute.

The advantage of PC polling is that it is about 15 times faster than the watchpoint mode and hence can keep pace with a higher frequency of events (function entry, exit, interrupt entry, exit, *etc.*). The disadvantage is that it does not allow for advanced triggers – it only allows reading the PC value. For example, we could not use PC polling alone to watch for a memory read or write to a specific memory location. A single iteration of the PC poll can be performed rather quickly; from Table 2, we see that it takes just $1.6\mu s$, which corresponds to 6.4 clock cycles of the application processor. It is rarely the case that interesting functions that we want to trace will take less than 6.4 cycles to execute.

However, implementing PC polling presents a practical challenge, namely, synchronizing the TDB and the application mote. This is because the TDB is reading the PC values while the mote is executing. To achieve this synchronization, we replace the mote's external crystal oscillator with a wire from the FPGA. Then, we modify how TinyOS configures the processor's clock module, as shown in Figure 7, so that the main clock (MCLK) on the processor is wired directly to the FPGA's clock as shown in Figure 8. This achieves synchronization of the TDB and the application processor.

In addition to reconfiguring the application processor's main clock (MCLK), we must also change the source of TimerA which requires a 1MHz clock signal, and TimerB which requires a 32kHz clock. The FPGA's 4MHz signal cannot be divided down to 32kHz, because LFXT1CLK and ACLK each have a maximum clock divider of 8. Therefore, we use the internal digitally controlled oscillator (DCOCLK) to provide a 32kHz signal to TimerB and connect TimerA to ACLK and remove the external crystal from the application processor board. A consequence of this is that DCOCLK

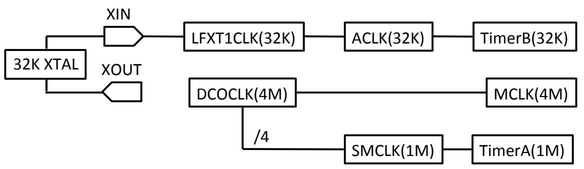

Figure 7. The setup of the MSP430 clock module used by TinyOS.

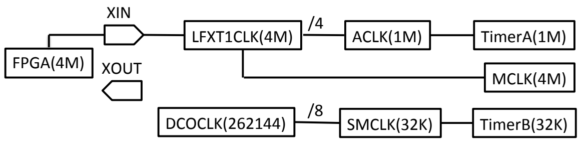

Figure 8. The modified clock module to synchronize the main clock (MCLK) with the FPGA clock.

Table 2. Time taken, in software and using the FPGA, to perform various operations through JTAG in the breakpoint, watchpoint, and PC polling modes.

Mode	Operation	Software (μs)	FPGA (μs)
Breakpoint	Test	13	3.8
	Read Addr.	42	12.2
	Resume	166	6.8
	Total	*221*	*22.8*
Watchpoint	Test	318	18.3
	Read Addr.	212	12.2
	Total	*530*	*30.5*
PC Polling	Read PC	55	1.6
	Total	*55*	*1.6*

cannot be turned off when the processor goes to sleep, because TimerB is responsible for waking the processor up. Another solution would be to connect the FPGA clock to the XT2 clock input pin. This would avoid removing the crystal oscillator and would allow TimerB to use the oscillator as its input, so the mote would not need DCOCLK in sleep mode. We chose the first approach for practical reasons, because the XT2 pin is physically less accessible than the crystal oscillator.

4 Experiments

4.1 Microbenchmarks

The objective of our microbenchmarking experiments is to evaluate the performance of the building blocks of AVEKSHA. In particular, we evaluate (a) how many clock cycles it takes for AVEKSHA to perform event monitoring each of the three modes – breakpoint, watchpoint, and PC polling, together with the individual components in each, (b) the accuracy of the energy monitoring by comparing it with measurements obtained using a Fluke multimeter as well as a dedicated power monitor from Monsoon Inc., and (c) the energy consumption of the TDB itself.

4.1.1 Time Taken in Each Monitoring Mode

Ideally, we would like to poll the PC or the watchpoint buffer at a rate sufficient to observe every single instruction executed on the mote. Unfortunately, the debug module on the MSP430 was only designed to be operated with a maximum frequency of 10MHz for the JTAG clock. One exception we have discovered empirically is that PC polling can be reliably clocked at up to 24MHz. Table 2 presents the effect this has on the time taken to complete basic polling operations. The software column shows how long operations take if only the MCU on the TDB is being used while the FPGA column represents the time operations take in the current FPGA implementation. The FPGA implementation is limited only by how fast the JTAG clock of the mote can be reliably driven. The table presents results in μs. TinyOS operates the main clock of the mote at 4MHz by default, so 1μs is equivalent to 4 clock cycles on the mote.

The breakpoint mode of monitoring comprises a test operation to determine if the mote has halted (which is done in a loop), a read address phase to collect at which instruction the mote halted, and a resume phase to restart execution on the mote. Likewise, the watchpoint mode has a test phase and a read address phase. The test phase here is more complex because it has to test if a new entry has been created in the JTAG circular buffer. In the watchpoint mode, a poll takes 122 cycles and this means that this mode can keep up with events if their sustained rate is less than 3.3×10^4 per second. For all of the applications that we have experimented with (which are a superset of the ones for which we provide results here), the rate of events, tasks, and application-level functions is lower than the above rate. However, if we include system-level entities (functions, events, and tasks), then this rate is occasionally exceeded. Finally, PC polling only requires a read PC operation that can be performed in less than 7 mote cycles. It is highly unlikely that functions of interest span less than 7 cycles and therefore PC polling, while not as expressive as watchpoint, is able to keep pace with functions that we want to monitor.

Table 3. Accuracy of the current measurements provided by AVEKSHA for fixed resistive loads, compared to values computed based on measurements with a Fluke multimeter.

Resistance	Current		Relative Error
(Ohms)	Computed (μA)	TDB (μA)	(Unitless)
179.71	18362.92	18483.56	0.007
218.55	15099.52	15193.74	0.006
560.8	5884.45	5807.06	0.013
991.4	3328.63	3314.46	0.004
4689.2	703.74	674.77	0.041
32610	101.20	92.34	0.088
55220	59.76	52.97	0.114
179360	18.40	16.09	0.125
266750	12.37	16.28	0.316

4.1.2 Accuracy of Power and Energy Monitoring

The objective of this experiment is to see if AVEKSHA can faithfully monitor the power draw in the static case (using a fixed resistive load) and when there are spikes in power consumption, which happen commonly in embedded systems, e.g., when the radio switches on. Table 3 shows the current consumption reported by the TDB for various resistive loads. For comparison, we measured the value of each resistor using a high-accuracy Fluke multimeter and computed the theoretical current consumption through it. As seen in the table, TDB's current measurement is within 10% of the computed value for current draws of 100 μA or above, while the error goes up for smaller current values. The accuracy of the current measurement can be improved further using techniques (which we have not implemented yet) such as better isolation between the analog and the digital components and better decoupling between analog components so that each has a closer-to-perfect grounding.

We also measured the power consumption reported by the TDB while attached to a TelosB mote running the `TestNetworkLpl` TinyOS application. It is important to note that the amplitude of the power consumption trace in this case will have a significant dynamic range due to various components on the mote changing power states during application execution. The TDB measurement of energy draw over a 1 minute period is within 3.2% of that given by a Monsoon power meter [11]. Since the spikes in energy draw are three orders of magnitude higher than the steady state case, this close result can only be achieved because TDB monitors the current spikes faithfully. For comparison, the static energy metering of iCount over a five decade range is accurate to 20% down to about 1 μA [12].

4.1.3 Power Consumption of TDB

It is important that the TDB itself consume a small amount of power because the typical usage scenario is the TDB coupled to the application processor board when the latter is deployed in the field. In this usage scenario, only the non-USB components on the application processor board are active. We find that our current prototype consumes a maximum of 55mW at a supply voltage of 3V. For comparison the TelosB mote with processor and radio active consumes about 49mW. So the TDB will approximately double the power consumption when the mote is fully active. It is possible to reduce this overhead by having the FPGA enter a sleep mode when the mote goes to sleep. This would involve triggering on the instructions that the application processor uses to set up a timer for going to sleep. Alternatively, the OS on the application processor could send a hardware signal to the debug processor on the TDB on a general purpose pin whenever the application processor is going to transition to a sleep state or immediately after waking up from it.

4.2 Application Setup

Our experiments use both TinyOS and Contiki applications without needing any extra programming effort since AVEKSHA is OS-agnostic by design. The only change to the OSes is the one required due to the change of the clock source that was required to synchronize the JTAG and application processor's clocks for PC polling (as shown in Figure 8). The clock initialization module is responsible for creating the main processor clock from the hardware clock and for wiring other internal hardware clocks to different sources. This module has to be changed in the OS for AVEKSHA to work with our clock setup.

We use two TinyOS applications `TestNetworkLpl` and `TestFtsp` and an object tracking application in Contiki. `TestNetworkLpl` uses the collection tree protocol to push sensor readings to a base station [13]. Each node samples the sensed value every 128 ms and, if there is no data to be read, goes to sleep. This is a typical application for sensor networks. We present a bug that was uncovered when we had AVEKSHA monitor all tasks in the watchpoint mode. `TestFtsp` is a time synchronization protocol [14]. The Contiki object tracking application takes light readings at each node and passes values that reach a threshold to the base station through a multi-hop routing protocol. This application wakes up to sample the light readings every 125 ms.

4.3 Watchpoints

4.3.1 Using States to Monitor Energy

One application for the TDB is to monitor various state variables. Monitoring state variables and transitions is useful because they can be correlated to power consumption, and can aid in understanding the behavior of applications (as

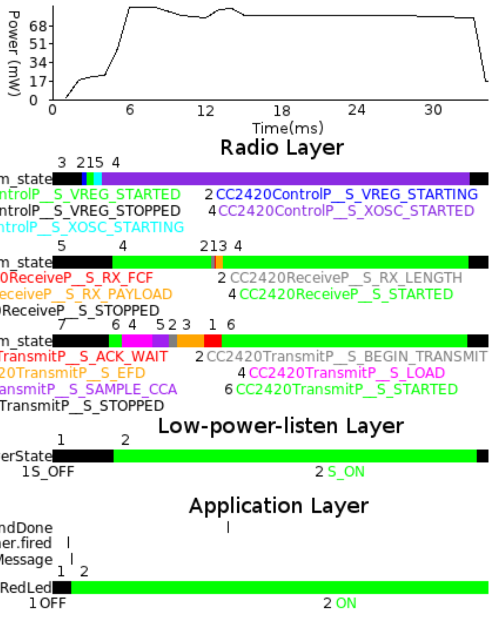

Figure 9. Watchpoint trace of states when sending a message in `TestNetworkLpl`, showing the application, low-power-listening, and radio layers. The number above each state's timeline corresponds to the numbering of the states under the timeline. For example, in the low-power listen layer, state 1 is S_OFF and 2 is S_ON; at the beginning the state is 1, then an extended period of state 2, followed by a return to state 1.

argued in Quanto as well [4]). This is particularly true in TinyOS, where the event-driven model encourages the use of explicit state machines.

In `TestNetworkLpl`, we have instrumented the application layer, low-power-listening layer, and the radio layer to monitor state changes. The instrumentation is simply to place a `nop` instruction which can be used as a trigger in the watchpoint mode. In the application layer, the beginning of every task and event handler is instrumented. In the low-power-listen layer, the state changes of interest are in the `RadioPowerState` module. This uses the state component interface in TinyOS and hence AVEKSHA inserts a `nop` whenever a function from that component is called. In the radio layer, state variables have the postfix `m_state`. We wrote a script that finds all assignments to these variables in the code and inserts a `nop` statement after the assignment.

Figure 9 shows a packet send that is initiated from the application layer when `Timer.fired` is triggered. The first step is to turn on the radio by starting the voltage regulator and oscillator, which is given by the state variable `CC2420ControlP_m_state`. The start up takes 1.6 ms, the duration of the `VREG_STARTING`, `VREG_STARTED`, and `XOSC_STARTING` states. The `CC2420TransmitP__m_state` shows the process of transmitting a message. The message transmission takes place during the states

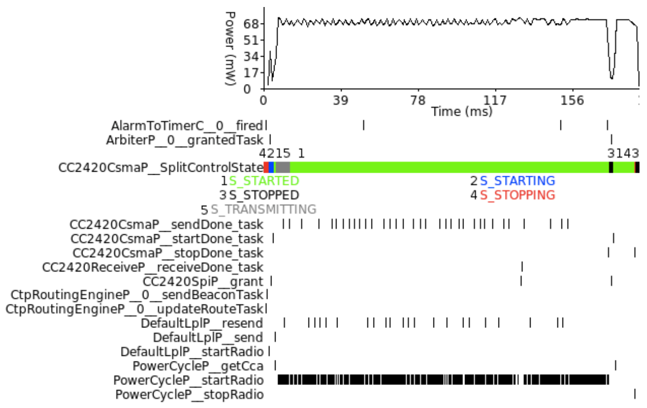

Figure 10. Watchpoint trace of task executions during a radio start event. The `PowerCycleP__startRadio` task is called over 3000 times due to a bug in the handling of the `CC2420CsmaP__SplitControlState`.

`S_BEGIN_TRANSMIT` and `S_EFD`. After the message is transmitted, the sender waits for an acknowledgment, which is shown in the `CC2420ReceiveP__m_state`. This variable shows the acknowledgment being received at 12 ms (the `S_RX_FCF` state) after which it is read off from the radio layer (the `S_RX_PAYLOAD` state). After this, the radio turns off at 32ms. A parameter of LPL controls the delay after receive and the default is set to 20ms which is verified by our experiment. This kind of low-level tracing of events in the stacks is useful for a developer wanting to get a detailed understanding of how a high-level function is accomplished (in this case, transmission of a message which requires an acknowledgment). Such an understanding can be used for performance tuning (speeding up some event in the time line, or reducing the amount of time spent in a particular state) or for energy optimization (knowing some energy-expensive state, reduce the amount of time the node spends in that state). This level of tracing would be very difficult to obtain through purely software means because of the fine-level of instrumentation that will be required, and correspondingly the high level of perturbation that will be caused to the normal execution of the application. On the other hand, AVEKSHA does not have to make tightly coupled changes to the hardware (the radio in this case), which are difficult to make and in some cases impossible when the hardware or the firmware is closed source.

4.3.2 *Using Tasks to Debug an Application*

The original objective of this experiment was to trace the collection tree protocol in the watchpoint mode. However, during the tracing, we observed some suspicious behavior that caused us to suspect that there was a bug in the low power listening layer of TinyOS. This was discovered by instrumenting all of the tasks in TinyOS for the `TestNetworkLPL` application. The nesC compiler in TinyOS creates a function called `SchedulerBasicP__TaskBasic__runTask` that contains a switch statement with a case for every task. By inserting a `nop` into each case, we can monitor every time any task is executed. We originally found that AVEKSHA was unable to

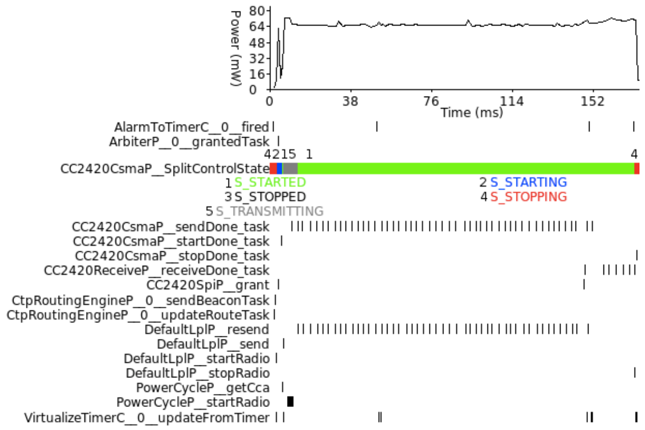

Figure 11. Watchpoint trace of task executions with the `startRadio` bug fixed.

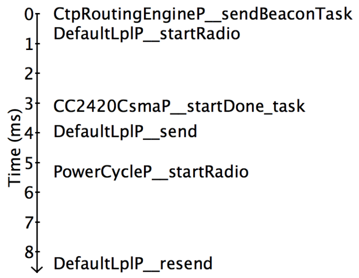

Figure 12. Execution timeline that causes task spinning.

keep pace with the rate of events that is generated after the mote is started up. Later, it turned out that this was due to a bug where some tasks were being repeatedly and unnecessarily re-posted.

Figure 10 shows a trace of the tasks shortly after the mote starts up. Three of the tasks (`PowerCycleP__startRadio`, `DefaultLplP__resend`, and `CC2420CsmaP__sendDone_task`) are stuck in a spin for more than a second after the mote starts. This implies that these tasks keep re-posting themselves and do not get any useful work done in each execution of the task. The spinning tasks are the result of the order of events that happen when the mote starts up. The timeline in Figure 12 shows the relevant events. First the collection tree protocol routing engine (`CptRoutingEngineP`) posts a task (`sendBeaconTask`) to send a beacon. This task results in the LPL module `DefaultLplP` posting a `startRadio` task. The radio is duly started (at around 90 ms) and the radio layer `CC2420CsmaP` sends a signal to `DefaultLplP` and `PowerCycleP` that the radio is started. At this time the beacon begins to be sent out and `DefaultLplP` attempts to send an initialization packet that will announce to other nodes what the duty cycle of this node is. Because the beacon is already being sent, the initialization packet cannot be sent and a timer is scheduled to start a resend. When the resend task `DefaultLplP__resend` fails (because the beacon is still being sent), it enters into a spin. Also, following the `DefaultLplP__send` task, `PowerCycleP` starts a task that is meant to start the radio in order to perform a CCA. If this task is unable to start the radio, it re-posts itself, again causing a spin. This is precisely where the bug lies. The radio has already been started and therefore this task should not re-post itself, but should return without doing anything. The bug is also not deterministic because if the `PowerCycleP` module had received the event before the `DefaultLplP` module, this bug would not have been seen. Eventually, when the sending of the beacon message and the initialization message are done, the radio is set off to sleep and `PowerCycleP's startRadio` operation succeeds. This can be seen from the CC2420 state of STARTED toward the end of the timeline (when it is started to do the CCA).

The buggy version of `PowerCycleP__startRadio` is shown first.

```
static inline void
PowerCycleP__startRadio__runTask(void) {
  if (PowerCycleP__SubControl__start() != SUCCESS)
  {
    PowerCycleP__startRadio__postTask();
  }
}
```

The undesirable effect of the bug is that it fills up the task queue (though a redesign in TinyOS 2.x limits this effect) and a task is being re-posted and invoked uselessly thus using up CPU resources.

The above is a real-case where the bug is activated. We hypothesize the following plausible application case where the bug will be activated and the `PowerCycleP`'s `startRadio` task will *never succeed and will keep spinning endlessly*. Consider an application that starts sending a message and shortly afterwards (after the radio has finished STARTING and entered STARTED state) turns off low power listening. A LowPowerListen interval of 0 indicates that low power listening should be shut off and the radio left on in receive mode. In this case, the task `PowerCycleP__startRadio` will never have the SUCCESS condition and will continue to spin until the LowPowerListen interval is again changed. We have confirmed that this happens when the following synthetic application is executed.

```
event void RadioControl.startDone(error_t err) {
  sendMessage();
  // 10 ms is sooner than message will
  // complete sending
  call Time.startOneShot(10);
}
event void Timer.fired() {
  call LowPowerListening.setLocalWakeupInterval(0);
}
```

To fix this bug, consider what happens to the state of the CC2420 radio (shown as `CC2420CsmaP_SplitControlState` in Figure 10). The function `PowerCycleP__SubControl__start()` tries to start the radio and tests the state of CC2420. If the state is STARTING it returns SUCCESS, if the state is STARTED it returns EALREADY, and if the state is anything else it returns EBUSY. Therefore, the simple fix to the task `PowerCycleP__startRadio` is as follows.

```
static inline void PowerCycleP__startRadio__runTask(void) {
  if (PowerCycleP__SubControl__start() != SUCCESS
  && PowerCycleP__SubControl__start() != EALREADY) {
    PowerCycleP__startRadio__postTask();
  }
}
```

Figure 11 shows that this fix indeed stops `PowerCycleP__startRadio` from spinning. Tasks no longer spin fruitlessly, but are successful in most instances.

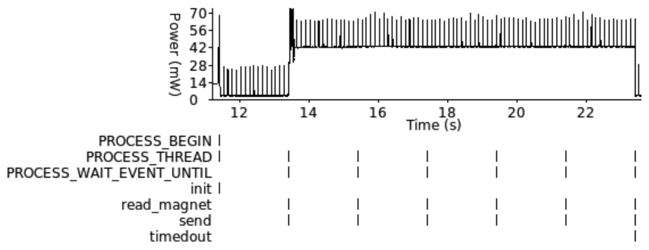

Figure 13. Watchpoint trace of application level functions and threads of a sender node in the Contiki tracking application.

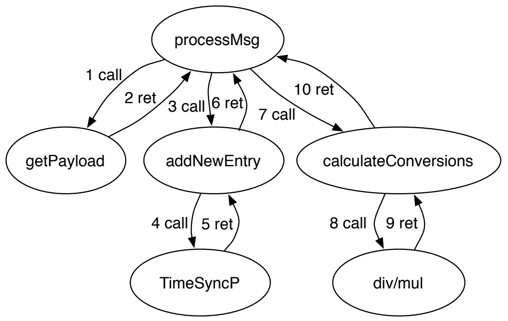

Figure 14. Call graph of the `processMsg` task in FTSP. There are multiple multiplication and division functions and time synchronization functions that have been lumped together as div and sync, respectively.

4.3.3 Processes in Contiki

An advantage of our approach over software tracing is that it is independent of the OS being used. Without any modification to the Contiki OS, the TDB is able to generate a trace of an application. In Figure 13, we show a trace of a sender node of a simple object tracking application called *LightTracker* [16], implemented in Contiki [17] version 2.4. LightTracker tracks a moving light source in a sensor network. There are two types of nodes present in the network: a *base station* and a set of *sender nodes*. A sender node periodically (every 2 seconds) collects light intensity using its light sensor and forwards it to the base station, possibly in a multi-hop manner, if the sensed value is above a threshold. The base station periodically checks the received samples and selects the node with the maximum light intensity. The selected node is considered to be the current position of the light source.

Unlike TinyOS, Contiki features the use of threads as a key design component. This reduces the need of maintaining explicit state machines in the code. In the sender application, we place a `nop` at the starting point of every thread command. `PROCESS_BEGIN` represents the creation of a thread and is performed once at time 11 seconds. `PROCESS_THREAD` is executed every time the thread is started. `PROCESS_WAIT_EVENT_UNTIL` is a blocking wait statement in the thread. We see that the power spikes correspond well with the `read_magnet` and send events.

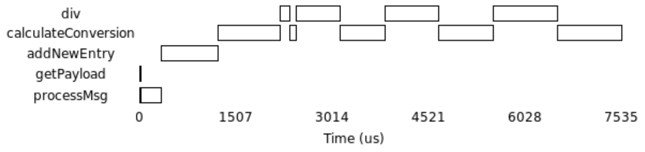

Figure 15. Trace of the different functions invoked in one execution of the `processMsg` **task.**

4.4 Profiling using PC Polling

Sampling the PC counter is a quick and non-intrusive operation. It does not have the flexibility of setting watchpoint triggers for specific conditions; however, it has the advantage of being able to measure events with greater timing accuracy than watchpoint polling. This is both because it is faster to take a sample of the PC counter and because it does not have to do buffer management.

A useful application of PC polling is for statistical profiling of an application, say to determine what parts of the code are most active. Raw PC polling data cannot directly give a profile of the number of times a function is called or the total time the function takes to execute, inclusive of the times of the nested functions that it calls.

However, AVEKSHA uses the raw PC polling data together with some statically generated information to get us the above measures. Figure 14 shows the static call graph for the `processMsg` task in `TestFtsp`. This task adds a synchronization method to a table by calling the `addNewEntry` function and then calling the `calculateConversion` function, which performs some fairly complex mathematical operations that necessitate repeated calls to div/mul (division/multiply). It would not be practical to store the result of every PC sample, because the rate of sampling is very high. AVEKSHA performs profiling by loading, at the beginning of execution of the application, a table of the start address of every function and interrupt into the FPGA. It then performs a binary search on every PC sampled to find the function that the PC belongs to. As described in Section 2, the search takes at most 10 cycles. The FPGA is clocked much faster than the application processor. The rated speed at which the FPGA can theoretically be clocked is 400 MHz (of course this would be for the simplest of operations). For our TDB, we clock the FPGA at 20 MHz and thus, 10 cycles at this speed completes before a single PC polling. Also, due to the parallelism inherent in the FPGA design, the pipeline shown in Figure 6 can be followed in AVEKSHA. Hence, the search can be supported without any increase of the cost of PC polling. Every time AVEKSHA detects that the C function has changed, it stores the new function together with a timestamp (the timestamp is automatically provided by PC polling). Figure 15 shows the results of sampling the PC. From this data and knowledge of the call graph, AVEKSHA is able to infer that this single call to the `processMsg` task took 7542 μs. It is also able to determine that the shortest call to div/mul took only 151 μs, a granularity that cannot be achieved with the watchpoint mode.

We are also able to know when the application processor is in sleep by monitoring for the address of the sleep instruction. This is possible because the address does not change once the application processor goes off to sleep and hence PC polling can detect this address. Using this we found that in a 30 second run of Ftsp, the application processor was active for 521 ms (giving a duty cycle of 1.7%). Of all the tasks, we found that the longest time was spent in the `processMsg` task - 121ms (or about 23% of the awake time).

4.5 Overhead of a Simple Software Profiler

Software profiling is used to collect and arrange different statistics about function calls in a program, such as the time spent in each function, how many times a function was called, *etc*. A prominent example of a software profiler is `gprof` [15], which is widely used in Unix systems. In this experiment, we measure the overheads that a software profiler can introduce in an embedded wireless node. We create a simple software profiler following the principles outlined in several papers [6, 1]. In it, profiling is performed by instrumenting with additional code, the entry and exit points of the functions that we are interested in. The additional code collects the time spent inside those functions.

For this experiment, we use the LightTracker object tracking application introduced earlier and instrument the function, `read_light()`, that is used by a sender node to collect the light intensity. Figure 16 shows the modified function. The newly added code for profiling is shown in red. Basically, the added code takes a time-stamp (lines 2 and 6) at the entry and the exit points of the function and stores their difference in a memory location (line 7). It should be noted that the time estimation is correct for a non-recursive function. For recursive functions, additional code is needed to keep track of the depth of the call in the recursion stack, which will incur additional overheads. Further, we only profile function entry and exit points and not other events of interest that AVEKSHA profiles, such as, point of an interrupt, data read from a peripheral, etc. From this perspective, our estimation of the overhead of the software-based profiler is conservative and provides a baseline for any implementation of a software-based profiler.

We simulate and profile LightTracker, both for the original and the instrumented version of `read_light`, using the Cooja [18] simulator for 500 seconds. In both cases, `read_light` was called 249 times. The average run-time for the regular version was 1,827 clock cycles per call. The version with code instrumentation for profiling spent 1,896 clock cycles on average. So, the additional code introduced an average overhead of 69 clock cycles per call. In comparison, in AVEKSHA, PC polling takes 1.6 clock cycles and watchpoint takes 30.5 clock cycles. In our simulation, a total of 6,52,109 calls for 158 different functions were reported. If we instrument each of those 158 functions, which would be the case for a complete profiler, the additional overhead will be significant. The instrumentation of `read_light` also increased the binary size by 201 bytes. For resource limited sensor nodes, the additional overheads, both in terms of run-time and binary size, are significant.

AVEKSHA avoids the overheads accociated with a software profiler because the application processor is relieved of the burden of performing profiling-related tasks, which are handled by the TDB instead.

```
static int read_light() {
  uint16_t start_time = (uint16_t)clock_time();
  SENSORS_ACTIVATE(light_sensor);
  int val = light_sensor.value(LIGHT_SENSOR_PHOTOSYNTHETIC);
  SENSORS_DEACTIVATE(light_sensor);
  uint16_t end_time = (uint16_t)clock_time();
  count_cycles[FID_READ_LIGHT] += end_time - start_time;
  return val;
}
```

Figure 16. A function with additional code (marked in red) for software profiling.

5 Discussion

The target mode of operation is the TDB coupled with the application processor board in field deployments. In this mode of operation, there is the consideration of where to store the logged events. For debugging, it is sometime sufficient to have a history of the last few events before some condition in the application was reached. For example, the OCDM of the MSP430 provides a history of the last 8 watchpoint events with this type of debugging in mind. If this is the case, the TDB's main processor can maintain a circular buffer of events in RAM, which would have a small enough memory footprint. If a larger history is required it is possible to store events into the TDB MCU's flash memory. For very long term storage an external USB storage host could be attached to the TDB or a compacted trace can even be sent to a computing platform over wireless communication .

An objective of AVEKSHA is to monitor the application processor without interfering with its execution. While the breakpoint mode is not suitable from this standpoint, breakpoints do have some useful potential for security and reliability. For example, breakpoints can be used to implement memory protection. The MSP430 has no memory protection and buffer overflow code injection exploits are known [19]. The attack works by injecting code into the stack and getting the processor to execute this code. Except for special cases, such as some boot loaders, the application processor should never need to execute code from RAM. All code is in flash which has a well-defined address range. By setting a breakpoint for an instruction fetch outside of this range, we can prevent code injection exploits, while allowing for the special circumstances such as boot loaders.

Our discussion of implementation details of AVEKSHA has been specific to the MSP430 MCU. This is a shortcoming of the current implementation, though the concepts apply to a broad class of embedded processors. The requirement for an OCDM within the processor, which can be accessed through the JTAG interface is fundamental to our design. The specific points of dependence on the MSP430 are the exact format of the instructions used as event triggers, the address ranges of the peripherals whose reads we are interested in, and understanding the state machine of the OCDM in the MSP430 which determines the sequence of instructions that AVEKSHA needs to send in the different modes (breakpoint, watchpoint, and PC polling). While breaking these dependencies and porting our solution to another embedded processor will require some effort (that we would roughly estimate as 40 man hours), we feel this is eminently doable.

6 Related Work

There are primarily three areas of work related to AVEKSHA, namely power measurement, hardware support for debugging embedded systems, and software-based debugging techniques for wireless sensor networks. We expand upon representative prior work in each of these three areas below.

Power measurement: The problem of estimating or measuring power (or energy) consumption has been addressed extensively in the context of various electronic systems. We restrict our discussion of prior work to techniques that specifically target sensor networks. Various sensor network simulators, such as POWERTOSSIM, AVRORA, and COOJA provide energy estimation capability based on pre-built power models of the target hardware platform. Measuring (as opposed to estimating) the power consumed by a sensor node is usually done using the so-called *sense resistor* approach (see Section 2). SPOT [3] is an energy meter for wireless sensor nodes that is based on the sense-resistor approach and uses a voltage to frequency converter to transform the voltage samples into an energy counter that can be read by the sensor node. iCount [12] is an energy meter design that targets sensor nodes that have a switching regulator. It provides energy metering capabilities at almost zero cost by just counting the cycles of the switching regulator. Quanto [4] builds on iCount by using regression models to obtain per-component energy consumption based on the aggregate measurement provided by iCount and also performs energy accounting to various application tasks through causal activity tracking. The Energy Endoscope project [5] uses a separate application-specific integrated circuit (called EMAP2), implemented using a micro-power fuse-based FPGA, to perform charge accumulation based on the sense-resistor method. Similar to designs such as SPOT, AVEKSHA provides energy measurement capability with a large dynamic range. However, in contrast to the above solutions, AVEKSHA has the added ability of correlating the energy measurement information with various temporal events logged by the board.

Sensor network debugging: Replay debugging is a well known technique for embedded systems [6]. Envirolog presents a software-only solution for recording events to flash memory [7]. Applications are annotated to indicate what should be recorded, which a preprocessor then turns into C code. During recording, 16 to 1024 bytes of RAM are used to buffer events which are then stored to Flash. FlashBox adopts a hybrid hardware/software approach to eliminate the bottleneck of writing to Flash [8]. In FlashBox, a second MCU and flash memory are added to provide dedicated recording. The compiler is modified to insert additional code at each location in the program that needs to be recorded. Events are directly sent via UART or GPIO to the dedicated MCU. The above techniques and other software solutions for event logging in sensor networks such as [1] have the disadvantage that they either perturb the timing behavior of the application, possibly suppressing some subtle bugs, or cause a large slowdown in application execution. A recent software solution [2] has focused on a specific kind of tracing (control flow tracing) and combines intelligent static analysis with run-time trace compression to decrease overhead. Nevertheless, this technique still requires applications to be instrumented to gather the tracing information. In contrast, AVEKSHA not only requires no modification to the application, but is also completely agnostic to the OS used.

Hardware support for debugging embedded systems: Real-time trace functionality has been implemented in many processor architectures. For example, the CoreSight Trace Macrocells provides hardware cores that can be added on as peripherals to an ARM-based system-on-chip to produce a cycle-accurate trace of execution. This includes the ability to collect and compress a large amount of trace data on chip and to transfer this data to a trace port interface unit, such as JTAG. The MSP430F1611 MCU used in the TelosB mote also has a limited built-in OCDM. The OCDM allows up to 8 triggers to be set and an instruction trace of 8 elements to be collected based on the triggers firing. This could be used, for example, to store the last 8 instructions executed before a trigger fired, or to record the last 8 triggers that fired. AVEKSHA goes beyond this by demonstrating that it is possible to provide CoreSight like functionality on a low end MCU such as the MSP430F1611 that is commonly used in wireless sensor nodes. Hardware designed to interface an OCDM to a host computer via the JTAG standard is often referred to as an In-Circuit Emulator (ICE) or In-Circuit Debugger (ICD), or more correctly, a JTAG adapter. Many ICE tools are available for the MSP430 processor family. An example is the open source GoodFET [20] tool. However, the GoodFET is only capable of operations involving reading and writing to flash. The debugging features of the OCDM (e.g., breakpoints, watchpoints, state storage) are not documented by Texas Instruments. To gain access to these features, we actually reverse-engineered parts of the integrated development tool IAR as it communicates through an ICD via JTAG [10]. Although IAR allows setting breakpoints and watchpoints, real-time trace generation of traces larger than the 8 entry state storage is not possible. AVEKSHA significantly enhances the debugging capabilities provided by IAR by eliminating this restriction. There exist several point solutions for hardware meant for tracking different kinds of control flow for the purpose of debugging, e.g., in [21], the authors design a hardware ASIC that monitors loops taken by tasks in a multi-tasking environment and performs this in a non-intrusive manner to the application.

7 Conclusion

In this paper, we have presented AVEKSHA, a hardware-software solution to the problem of tracing events at runtime in an embedded wireless node, without slowing down the application. AVEKSHA can trace a variety of events, such as, particular PC addresses, reads from peripherals, entry and exits from tasks and interrupt service routines, and arbitrary user-defined events. We have shown through two applications in TinyOS and one in Contiki that such tracing is useful for profiling the execution times of different tasks and event handlers. While the watchpoint mode of operation is capable of capturing a practically limitless variety of events, it cannot keep pace with events that occur more frequently than 122 clock cycles on a sustained basis. The PC polling mode of operation is restricted in the kinds of events that it can detect, but being faster it can keep pace with events that occur at a rate less than every 7 clock cycles. Our profiling of tasks uncovered a performance bug in the low power listen radio module of TinyOS, which we were able to fix. Further, AVEKSHA has the ability to do energy monitoring over the μA to mA range and coupling it to execution regions between two events of interest.

This work points to the feasibility of tracing a wide variety of events of interest in a low-cost and non-intrusive manner while the embedded node is deployed in the field. We do not have to rely on expensive and custom-built hardware to achieve this. The events provided by AVEKSHA can be used by a variety of existing and yet-to-be-developed solutions, such as replay-based debugging, performance profiling, and energy monitoring.

8 References

[1] M. Wang, Z. Li, F. Li, X. Feng, S. Bagchi, and Y.-H. Lu, "Dependence-based Multi-level Tracing and Replay for Wireless Sensor Networks Debugging," in *LCTES*, 2011.

[2] V. Sundaram, P. Eugster, and X. Zhang, "Efficient diagnostic tracing for wireless sensor networks," in *SenSys*, 2010.

[3] X. Jiang, P. Dutta, D. Culler, and I. Stoica, "Micro power meter for energy monitoring of wireless sensor networks at scale," in *IPSN*, ACM, 2007.

[4] R. Fonseca, P. Dutta, P. Levis, and I. Stoica, "Quanto: Tracking energy in networked embedded systems," in *OSDI '08 USENIX Association*, 2008.

[5] T. Stathopoulos, D. McIntire, and W. J. Kaiser, "The energy endoscope: Real-time detailed energy accounting for wireless sensor nodes," in *IPSN*, 2008.

[6] H. Thane, D. Sundmark, J. Huselius, and A. Pettersson, "Replay debugging of real-time systems using time machines," in *IPDPS*, 2003.

[7] 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," in *INFOCOM*, apr. 2006.

[8] S. Choudhuri and T. Givargis, "Flashbox: a system for logging non-deterministic events in deployed embedded systems," in *SAC Symposium on Applied Computing*, ACM, 2009.

[9] "Green hills software inc." http://www.ghs.com/.

[10] "IAR Embedded Workbench for TI MSP430." http://www.iar.com.

[11] "Monsoon inc. power monitor." http://www.msoon.com/LabEquipment/PowerMonitor/.

[12] P. Dutta, M. Feldmeier, J. A. Paradiso, and D. E. Culler, "Energy metering for free: Augmenting switching regulators for real-time monitoring," in *IPSN*, 2008.

[13] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis, "Collection tree protocol," in *SenSys*, ACM, 2009.

[14] M. Maróti, B. Kusy, G. Simon, and A. Lédeczi, "The flooding time synchronization protocol," in *SenSys*, ACM, 2004.

[15] S. Graham, P. Kessler, and M. Mckusick, "gprof: a Call Graph Execution Profiler," in *SIGPLAN*, ACM, 1982.

[16] M. S. Hossain, A. B. M. A. A. Islam, M. Kulkarni, and V. Raghunathan, "μSETL: A Set-based programming abstraction for wireless sensor networks," in *IPSN*, 2011.

[17] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki-a lightweight and flexible operating system for tiny networked sensors," in *EmNets*, 2004.

[18] F. Osterlind, A. Dunkels, J. Eriksson, and N. Finne, "Cross-level sensor network simulation with cooja," in *LCN '06*, pp. 641–648, 2006.

[19] T. Goodspeed, "Msp430 buffer overflow exploit for wireless sensor nodes." http://travisgoodspeed.blogspot.com/2007/08/machine-code-injection-for-wireless.html, August 2007.

[20] T. Goodspeed, "Goodfet." http://goodfet.sourceforge.net, May 2010.

[21] K. Shankar and R. Lysecky, "Control Focused Soft Error Detection for Embedded Applications," *Embedded Systems Letters, IEEE*, vol. 2, no. 4, pp. 127–130, 2010.