Software Instrumentation for Failure Analysis of USB Host Controllers

Antonio Sabatini, Nathan Jarus, Pratik Maheshwari, and Sahra Sedigh
Department of Electrical and Computer Engineering
Missouri University of Science and Technology
Rolla, Missouri, 65409–0040
Email: {ajs5gd, nmjxv3, prm8c7, sedighs}@mst.edu

***Abstract*—Failures caused by electrostatic discharge (ESD) compromise the reliability of embedded systems. Peripherals such as the universal serial bus (USB) are particularly vulnerable, as isolating them to avoid electromagnetic interference would defy their purpose - facilitating communication with and/or by the embedded system. Better understanding the propagation of failures that result from ESD would facilitate defensive development of hardware and software for embedded systems, but is hampered by the lack of non-invasive and lightweight instrumentation techniques. This paper proposes the use of software instrumentation for monitoring the reaction of the USB peripheral of an embedded system to ESD. It describes our efforts towards detection and root cause analysis of ESD-induced failures - correlating changes in the operation of the peripheral with the specific pin subjected to ESD. The work described is intended as proof-of-concept for the development and use of (in situ) software instrumentation for lightweight acquisition of data that can be used for runtime failure analysis and actuation of self-healing mechanisms, as well as postmortem statistical analysis of system reliability, availability, and survivability.**

***Index Terms*—software monitoring, software instrumentation, electromagnetic immunity, failure analysis, USB**

I. Introduction

Electrostatic discharge (ESD), defined as the momentary flow of charge between two objects at different electrical potential, is a common cause of damage to and failure of embedded systems and their peripherals. Preventing this damage and failure, and mitigating its effects when it inevitably occurs, is most effective when the root cause is known and considered. The increasing complexity and shrinking form factor of embedded systems significantly complicates root cause analysis of failure, especially through methods that rely on hardware instrumentation for collecting data about system operation and failure [1].

The broad scope of the effects of ESD, which extends beyond hardware and causes software failures and data corruption, necessitates the use of monitoring techniques that can capture and reveal the internal 'state' of the embedded system as reflected in flags and registers. In one experiment, we exposed a camera to ESD. The display screen on the back of the camera became distorted to the point where no text or image could be seen; the screen would light up, but was blank. Further investigation (using a software-based monitor) revealed that the blank display was the result of a software failure caused by ESD. Determination of this root cause would not have been possible with hardware probing alone, in part because the failure data yielded by such probing lacks the semantic context offered by state capture.

Software-based instrumentation, where a program is used to monitor and record the state of the embedded system offers a suitable compliment to hardware-based techniques. A major benefit of software instrumentation, when correctly implemented, is that it is non-invasive and does not alter the environment being monitored. The ultimate goal of our work is the development of software instrumentation algorithms and techniques that are lightweight, accurate, and portable (across platforms); and can be easily customized to monitor different parts of an embedded system.

The work presented in this paper - software instrumentation of the USB host controller - serves as a preliminary effort and proof-of-concept. Extension of this work to concurrent monitoring of several peripherals is in progress. We seek to address a fundamental challenge in instrumentation of embedded systems: eliminating both probing hardware and software overhead. We expect the insights gained to a) facilitate identification of areas of an embedded system; e.g., specific pins; that are most vulnerable to the effects of ESD; b) enable root cause analysis of failures caused by ESD; and c) yield better understanding of hardware coupling paths of ESD.

II. Related Work

System monitoring techniques fall into one of three categories: hardware-based, software-based, or hybrids of the two. Regardless of the category, a fundamental requirement for any monitoring technique is that it not interfere with regular operation of the system. Desirable features include low overhead, low cost, accuracy, survivability (the ability to continue monitoring despite the failure of parts of the system), and high resolution. In this section, we elaborate on the success of existing monitoring techniques in achieving these design objectives and compare and contrast them to each other and to our proposed approach. We presented a more detailed review of existing techniques, with special focus on software, in a recent paper [2].

In monitoring the effects of electromagnetic interference, hardware-based monitoring techniques far outweigh others. The general hardware-based approach is to subject a system to

electric (E) and magnetic (H) fields and observe the system- or chip-level reaction, with the goal of identifying resulting faults [3]–[5]. The main shortcoming of these methods is that they are capable of detecting only the ultimate effect of the failure; e.g., LCD disturbance or short circuits. Detection of the root cause of the behavior, failure propagation, and similarly insightful information is not possible. Furthermore, these methods require direct access to the system by an external measurement device, and embedding them to allow runtime monitoring is impractical. Another shortcoming of hardware-based techniques is the cost of the measurement devices required.

Software techniques typically involve the generation of *softprobes* to monitor hardware-independent information with the intention of adding minimal to zero additional overhead on the system. Softprobes are scripts that are triggered in response to predefined events; e.g., changes to the value of a register. The additional computations required can affect the system performance and lower the accuracy of monitoring, as the overhead based by the softprobes alters the system state.

This specific shortcoming - high overhead - is addressed by Callanan et al. [6], whose technique allows the user to set a predetermined threshold for the overhead. HCSM then maximizes the number of events that it can monitor while remaining under the target overhead. The authors apply the technique to memory leak detection and bounds-checking, both of which have relatively predictable effects on a system. The unpredictable nature of the effects of ESD limits the utility of techniques that are subject to hard constraints on overhead. Our proposed approach has no such constraint, but we make every effort to minimize the overhead incurred.

In addition to overhead, a second challenge faced by software-based instrumentation is hardware failure: the software application becomes useless if the system crashes. Our approach is vulnerable to this challenge, which we are in the course of addressing. The work of Li et al. [5] - a hardware-based approach that involves an application that modifies the functionality of the system in the event of component failure, allowing for monitoring to continue. The article focuses on the Xilinx XUPV2P board that is used to capture audio files using different filter regions [5]. The goal of their work is to demonstrate the usefulness of a partially-reconfigurable monitor module. When a failure happens, instead of a full system crash, the monitoring tool will determine what can remain operational, and salvages any system functionality that can be maintained. Significant redundancy is required for such reconfiguration to be possible, and generalization of the technique from one platform to another is rarely possible.

A hybrid hardware and software monitoring technique is explained in [7], where the authors created a device that runs on an FPGA board and plugs into a PCI/PCI-X socket on the system. This separate device monitors the data bus to which it is connected and can detect other components on the same bus. The traffic on this shared bus is analyzed to determine whether any component connected to it has failed. Based on the failure detected, the monitor determines a recovery action and sends the appropriate command over the bus. The recovery actions can vary from rebooting the failed component, to disconnecting a failed peripheral. Decoupling the monitor from the system reduces overhead, but attaching the monitor is invasive. The efficacy of this technique is limited, as a very limited set of flags were monitored. Open source software is the foundation for the hybrid approach proposed by Cataliotti et al. for measuring power consumption [8]

Our earlier publication on this topic described a software-based technique for monitoring the SD card peripheral of a S3C2440 development board [2]. In preparation for this case study, we investigated a number of existing software-based tools for monitoring an embedded system subjected to ESD. Three of them are noteworthy: usbmon, Keil, and devmem. The first tool, *usbmon* [9], was determined to be incapable of collecting sufficient data for fault localization, primarily because low-level information about the hardware could not be obtained. The debugger that is built into Keil [10], an integrated development environment, was found to have excessive overhead that limited the effectiveness of the sampling. We determined that the JTAG adapter was the cause of the problem, as its maximum clock rate is 1 MHz. The third tool, *devmem* [11], required the use of multiple scripts for instrumentation. The resulting overhead and invasive probing interfere with the system's performance, increasing the difficulty of simulating a test environment that replicates a realistic environment.

Our previous work [2], which analyzed the effect of ESD on the secure digital (SD) peripheral, focused on determining whether existing tools could suffice for collecting information that would allow for root cause analysis of failures caused by ESD. The conclusion we drew was that a purpose-built tool was necessary, due to both overhead and reliability concerns with existing tools. The work presented in this paper applies the lessons learned in developing simple scripts whose lean design increases the reliability and decreases the overhead associated with monitoring. Throughout design and development of the monitoring technique, scalability to additional peripherals and generalization to other platforms were key concerns.

III. Proposed Approach

When an embedded system experiences ESD, its effects can flip bits in a data stream, change register values, cause timing issues, or corrupt data; leaving an error that may be observable at the software level. By observing and attempting to predict these errors, we can detect ESD and respond to it in software to create a more reliable system. Our approach to detection of ESD involves modification of the software drivers for the peripheral being studied, so they can be utilizes to log detailed information on hardware and software errors.

Additional software generates continuous data transfer to and from a peripheral, keeping the peripheral active. A high-voltage ESD simulator such as a transmission line pulser (TLP) is connected to an H-field or E-field probe and ESD is injected on the device under test at predetermined voltages.

The probe is used to focus ESD injection on different pins and busses used by the peripheral. Errors generated by the software are then studied to determine how the peripheral is vulnerable to ESD.

The experiments in this paper focus on the USB host interface of the FriendlyArm Mini2440 embedded development board, which has a Samsung S3C2440 ARM9 processor. The host interface conforms to Open Host Controller Interface (OHCI) specifications [12]. The system is configured with a custom compiled Linux kernel, version 2.6.29, retrieved from the FriendlyArm website [13].

Initially, the softprobe was implemented as a Linux kernel module that sampled arbitrary registers. The objective was to determine the hardware state of various subsystems from control and status registers. Exposure to ESD changes the state of the hardware, causing status registers to change. By observing these changes in software, it can be determined if and how the system has experienced ESD. The registers to be sampled are specified in a configuration file, as shown in Figure 1. Continual monitoring of these registers captures the manifestation of errors caused by ESD.

The register reader kernel module is inserted, and a user level program, myregrw, which continually instructs the module on the registers to be read (based on the configuration file), is launched. ESD is injected and the results are recorded for later analysis. For the USB host interface, watching the group of OHCI standard control and status registers at addresses 0x49000000 through 0x49000014 provides an indication of how the USB hardware state is influenced by ESD. Table I shows a number of failure states for the HcInterruptStatus register.

This initial approach failed, for two reasons. Firstly, achieving a register sample rate sufficiently high for observing ESD-induced errors proved to be difficult, due to data logging bottlenecks. The software was empirically determined to be capable of sampling one register approximately 342 times per second. Assuming that the board executes one instruction per cycle and that the register is reset after 1 instruction, this gives a worst-case probability of $\frac{342}{400*10^6} * 100 = 0.000856\%$ of observing an error in a status register before it is reset. Even though this is worst-case analysis, the probability would have to be improved by four orders of magnitude to approach even a 10% observation rate. Secondly, the Linux drivers for the hardware modify the control and status registers for devices as well, so changes cannot be uniquely attributed to ESD. As these drivers are given priority over user code, it is nearly impossible to read the register values after the occurrence of the error and before the hardware driver resets the registers.

Our second approach involved modification of the source code for the hardware drivers themselves, as shown in Figure 2. Drivers for the peripherals being studied were built as kernel modules, instead of being built directly into the Linux kernel. Building drivers as a kernel module places them into a separate module file that can be loaded manually, instead of requiring the entire kernel to be rebuilt and reinstalled after each modification to the drivers.

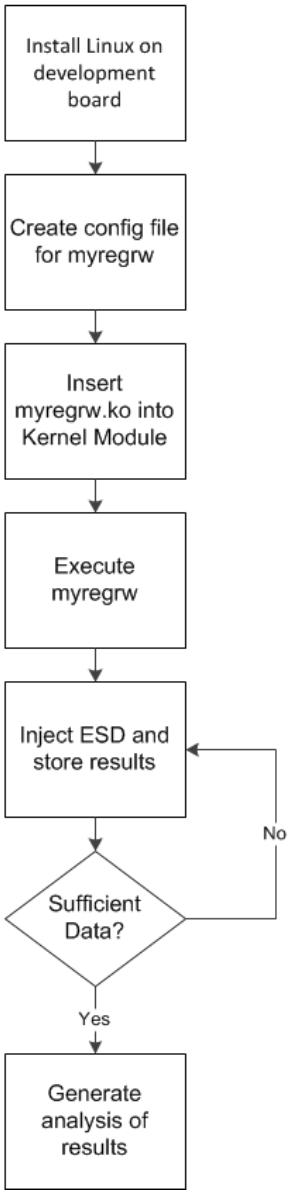

Fig. 1: Initial Approach: Register Sampling

Next, the drivers are modified to log details of software failure. The debug configuration available in most drivers, which gives some detail about driver state, is enabled. As the default debug messages are not detailed enough for the required analysis, the driver is then modified to record more data to the system logs, via calls to printk(). This approach is the least invasive to the driver software, since it requires only trivial modifications. Data is logged to pinpoint the failure within the communication protocol and hardware drivers.

For the USB host controller, two types of information are logged: information from data structures in the USB driver code, and execution paths through the driver. Some data structures are mapped to specific memory addresses that correspond to registers for the USB host controller; these registers will be logged. In the OHCI drivers, the struct ohci_regs defines

Value	Failure State	Comments
0x00000001	SchedulingOverrun	Hardware register value may change unexpectedly
0x00000010	UnrecoverableError	System has experienced an unexpected error
0x00000020	FrameNumberOverflow	Hardware register value may change unexpectedly
0x00000030	RootHubStatusChange	Status change may be caused by ESD

TABLE I: HcIntteruptStatus Failure Modes

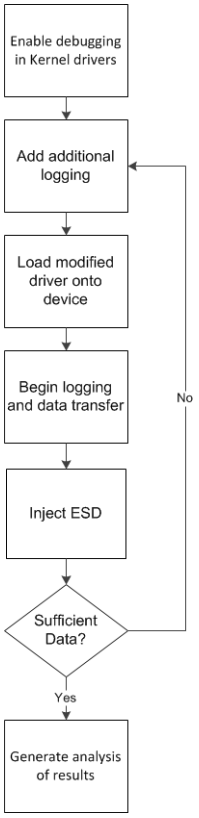

Fig. 2: Second Approach: Driver Modification

these mapped registers. The host controller driver consists of several functions that are called when certain events occur; for example, ohci_irq is called when an IRQ occurs for the host controller. Logging which function is being called gives a rudimentary idea of the operational state of the hardware.

After modification, the driver module is recompiled and downloaded to the device and the driver module is inserted into the kernel. Then, software is run that repeatedly transfers known data patterns to and from the peripheral being studied. This keeps the peripheral active during the testing. As well, it gives complete control over the software stack, eliminating uncertainty over what bit patterns are being transferred to or from the device. In our tests, we simply copied a file from a USB drive to the device, causing communication to and from the USB drive to the Host Controller.

Then, noise is injected using the TLP on the pins and busses used by the peripheral. The data from the kernel drivers is logged to one file using the Linux log processing tool syslog, and the ESD pulse timestamps are logged to a separate file

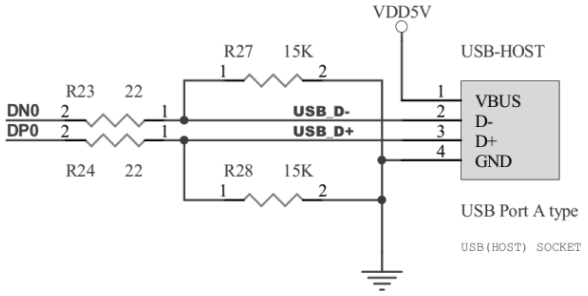

Fig. 3: USB Host Hardware Schematic

for later analysis.

For the USB host hardware, ESD is injected on wiring between the board's USB port and CPU pins DN0 and DP0, which drive the USB Data+ and Data- pins through a resistor network as shown in Figure 3. Injection is also performed after the resistor network directly on the Data+ and Data- pins.

Using the timestamps for each ESD injection, data logged from the drivers can be correlated with an ESD injection of a certain voltage and pulse width. This data can then be used for a wide variety of applications in both hardware testing and error recovery; examples include determining what hardware needs reliability improvements and software recovery from ESD.

IV. Analysis

A. Data Format

The data generated for each ESD injection is split into individual log files per injection. As well, several logs of normal operation are captured to give a basis for predicting or detecting ESD.

Every call to a function in the host controller driver generates a log entry containing a timestamp, the name of the host controller driver function, and a dump of the values of all the registers of the host controller at the time of the function call.

B. Individual Log Analysis

The results from an individual log, whether that log contains events from the hardware being exposed to ESD or not, can be seen as a series of states $S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow \ldots$ where each state represents the data logged from one driver function call as the pair (function call, register values). This gives a linear progression of the driver operation.

However, it is more useful to represent driver operation as a digraph where the nodes are unique states in the log, and edges are transitions between states, as in Figure 4. We can convert the chain of original states S into a set of locally-unique states

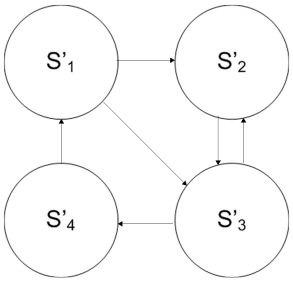

Fig. 4: Graph of Abstract System States

(unique within the log) S' and a list of state transitions in the order they are performed.

Locally unique log states are generated by comparing the function call and register values between the two, ignoring the timestamps of the states and also the HcFmRemaining and HcFmNumber registers, as those change every state and add an additional layer of complexity to the analysis. If two states match, they both refer to the same unique state. As states are coalesced into locally unique states, the call order of the unique states is noted.

C. Combined Log Analysis

In order to be able to draw conclusions from the entirety of the data present, we must now generate a global state graph containing states S'' such that, if we let L_i be the unique states S' of log i, $S'' = L_1 \cup L_2 \cup \cdots \cup L_n$. Along with this global state graph, each log's state transition list must be updated to reflect states in S'' instead of the states in L_i.

Performing the union of two locally-unique state sets is somewhat complicated, since certain registers can have different values every time the host controller driver is started. These registers are HcPeriodCurrentED, HcBulkCurrentED, HcHCCA, HcControlHeadED, HcControlCurrentED, and HcBulkHeadED. Since these registers hold pointers to dynamically allocated memory regions, they change to point to different regions when the driver restarts, but a difference in one of their values between states from different logs does not necessarily mean those states are not the same global state. However, these registers cannot be ignored entirely, since they do sometimes change in normal operation of the driver.

We therefore can write two comparison routines: weak-compare, which compares states, ignoring registers that may change between logs, and strong-compare, which compares every register value between states, excluding HcFmNumber and HcFmRemaining.

In order to prevent several unique states from one log being assigned to the same globally unique state, each globally unique state can only be associated with one unique state from a log file. To implement this, a globally unique state S''_i is represented as an array of log unique states, one for each log having a state weakly comparing to all the other states in S''_i, or $\varnothing$ if no state from that log corresponds to S''_i. Figure 5 diagrams how different unique states from log files are combined to represent globally unique states.

D. Interpreting the Globally Unique State Graph

Once the globally unique state graph is generated, it's possible to directly compare state transition patterns between logs where the board was not exposed to ESD and logs where the board was. We can thus determine the possible state transitions between a known-good state to a known-bad state, giving a starting point for root-cause analysis.

To find these states, we create a set of all state transitions in the non-ESD-exposed logs, N, and another set of all state transitions in the logs from ESD exposure, E. The difference of these two sets, $D = E - N$, contains all state transitions that may be a result of ESD exposure. The first state transition in each error log that is also in D is thus the state transition where abnormal behavior in the driver is first observed. Knowing which transitions initially indicate ESD gives the ability to detect when ESD injection occurs, as well as a starting point for root-cause analysis.

V. Results

Our results are based on 4 baseline logs without ESD exposure and 18 logs with ESD exposure. Testing was performed with a variety of voltages and ESD injection probes. From these logs we derived a globally unique state graph with 78 vertices; 20 of those states were present in logs without ESD exposure, and 74 in logs with ESD exposure. The baseline logs had 29 state transitions, and the ESD exposed logs 179. These data indicate that a marked difference between hardware operating exposed to ESD and hardware operating normally can be observed in software.

The error logs contained 10 unique initial state transitions not present in the baseline logs. 8 of them transition into states only found in ESD exposed logs, indicating that many exposures to ESD result in the hardware transitioning to an unexpected state. 2, however, transition to a state also present in the baseline logs, implying that certain ESD exposures may be able to cause unexpected transitions between otherwise valid states.

Further analysis of the results presented here are needed to determine how the effects of ESD exposure affect the hardware. As well, additional testing, both for normal and ESD exposed operation, needs to be performed to verify that our results are applicable to the driver operating in general, and not just the driver's operation as recorded in the analyzed logs.

VI. Potential Applications

The relationship between errors and ESD can be reversed, allowing predictions about what pins have received ESD based on errors the software experiences. As a result, using only software, components that have received ESD can be identified, either for replacement if the goal of the experiment is to repair hardware, or for improvement, if the goal is to reduce the effects of ESD on a peripheral.

Specific software errors or sequences of errors can be used to predict that the hardware has experienced an ESD discharge. Using this information, software can be written to recover from ESD, possibly without having to entirely disable and re-enable

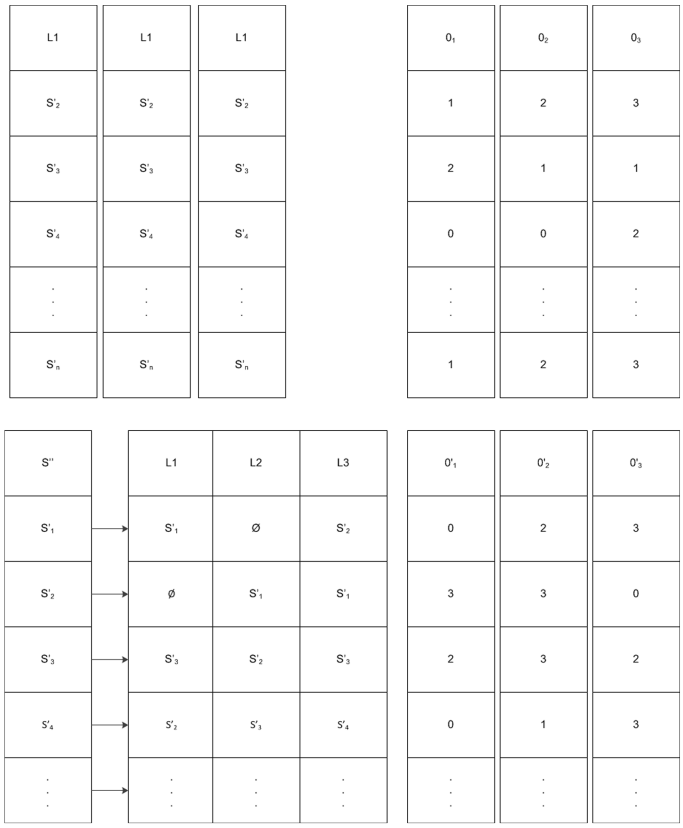

Fig. 5: Graph of Abstract States

the peripheral. Software may also be able to compensate for the effects of ESD, allowing operation to continue in adverse environments, albeit at the cost of reduced performance and more software overhead.

VII. Conclusion

This paper discussed the process of porting a method of testing ESD on a SD card using software to testing USB host hardware. The goal from this work is to improve the technique and allow the software to identify and self correct errors that were caused from interference. The methodology behind the software modifications will lead to the improvement of the reliability of the system. This methodology also focuses on reducing additional overhead on the test system and using software to gather data for statistical analysis.

References

[1] S.-Y. Yuan, Y.-L. Wu, R. Perdriau, and S.-S. Liao, "Detection of electromagnetic interference in microcontrollers using the instability of an embedded phase-lock loop," *IEEE Transactions on Electromagnetic Compatibility*, vol. PP (early access preprint), no. 99, pp. 1–8, 2013.

[2] P. Maheshwari, T. Li, J. Lee, B. Seol, S. Sedigh, and D. Pommerenke, "Software-based analysis of the effects of electrostatic discharge on embedded systems," in *IEEE Computer Software and Applications Conference (COMPSAC)*, July 2011, pp. 436 – 441.

[3] K. Wang, J. Koo, G. Muchaidze, and D. Pommerenke, "ESD susceptibility characterization of an EUT by using 3D ESD scanning system," in *International Symposium on Electromagnetic Compatibility EMC'05*, vol. 2, August 2005, pp. 350–355.

[4] G. Muchaidze, J. Koo, Q. Cai, T. Li, L. Han, A. Martwick, K. Wang, J. Min, J. Drewniak, and D. Pommerenke, "Susceptibility scanning as a failure analysis tool for system-level electrostatic discharge ESD problems," *IEEE Transactions on Electromagnetic Compatibility*, vol. 50, no. 2, pp. 268–276, May 2008.

[5] Z. Li, J. Xiao, B. Seol, B. Lee, and D. Pommerenke, "Measurement methodology for establishing an IC ESD sensitivity database," in *Asia-Pacific Symposium on Electromagnetic Compatability (APEMC)*, April 2010, pp. 1051 – 1054.

[6] S. Callanan, D. Dean, M. Gorbovitski, R. Grosu, J. Seyster, S. Smolka, S. Stoller, and E. Zadok, "Software monitoring with bounded overhead," in *IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008.*, april 2008, pp. 1 –8.

[7] J. Wang, K. Sun, and A. Stavrou, "Hardware-assisted application integrity monitor," in *Hawaii International Conference on System Science (HICSS)*, January 2012, pp. 5375 – 5383.

[8] A. Cataliotti, V. Cosentino, D. Di Cara, A. Lipari, S. Nuccio, and C. Spataro, "A PC-based wattmeter for accurate measurements in sinusoidal and distorted conditions: Setup and experimental characterization," *IEEE Transactions on Instrumentation and Measurement*, vol. 61, no. 5, pp. 1426 –1434, may 2012.

[9] P. Zaitcev, *The usbmon: USB monitoring framework*.

[10] "KEIL embedded tools," http://www.keil.com/.

[11] "Devmem2," http://sources.buildroot.net/devmem2.c.

[12] "Open Host Controller Specifications for USB," http://download.microsoft.com/download/1/6/1/161ba512-40e2-4cc9-843a-923143f3456c/ohci_11.pdf, accessed: 27/02/2013.

[13] "Downloads - FriendlyArm," http://www.friendlyarm.net/downloads, accessed: 27/02/2013.