A Software Method for Detecting Electrostatic Discharge in USB Host Controllers

Antonio Sabatini, *Member, IEEE* Nathan Jarus, *Member, IEEE* Pratik Maheshwari, *Member, IEEE* and Sahra Sedigh, *Member, IEEE*

***Abstract*—Understanding the effect electrostatic discharge (ESD) has on embedded systems and their peripherals is useful for both hardware and software design. Manufacturers must design hardware that can withstand ESD during normal operation. They may also wish to know which components on a defective board have been damaged by ESD in order to repair the system. Software developers may want to ensure that the software controlling these peripherals can quickly recover from ESD without crashing the system.**

In order to instrument commercially produced hardware, we develop a software-based ESD monitoring methodology requiring no special hardware on the embedded system. This methodology can be applied to any embedded system peripheral and allows the system to operate normally while the monitoring software is running. We apply our methodology to a USB host controller on an embedded system and explain the significance of the registers we monitor. Furthermore, we develop a classifier capable of determining whether or not the system is experiencing ESD based on its current state.

***Keywords*—Software instrumentation, electrostatic discharge, android models, non-invasive, deterministic data, failure analysis, USB peripheral**

I. Introduction

As embedded systems become smaller and smaller, they become more vulnerable to physical events and thus more difficult to make reliable. Electrostatic Discharge (ESD) is a major cause of this unreliability, since smaller components require a smaller electrical charge to experience an ESD event. The effects of these events on the software running on the embedded system are not well understood yet. In order to understand them, we must observe how the hardware effects of ESD appear to the software controlling that hardware.

Some ESD events do not cause permanent damage to the hardware but instead cause software glitches that appear random and unexpected to the system user. Hardware-level analysis methods struggle to monitor these types of events; it is difficult to associate a user-observed event with a specific physical event. Additionally, component miniaturization is increasing the difficulty of monitoring all traces on a board for ESD. Even if all the traces could be monitored, analyzing how the ESD coupled to the hardware and what effects it had on various components is challenging. Finally, while invasive hardware testing might be feasible on a development board, testing on consumer hardware not designed for such testing is much more difficult.

In order to avoid these issues with hardware ESD analysis, we propose a low-level software analysis method. Software instrumentation allows hardware that cannot be physically probed to be observed during ESD events. However, many software analysis techniques focus on high-level faults instead of hardware failures. Other software approaches study low-level failures, but run on the bare metal, preventing the embedded system from being used as it was intended. Our approach uses modified hardware drivers to allow a fully-functioning system to be monitored for ESD events. In our work, we consider small-scale ESD events that do not cause permanent system harm.

With software instrumentation, we are able to observe the changes in system state caused by ESD. From this data, along with data concerning normal system operation, we can construct software capable of predicting when a system is experiencing ESD. These results have several applications in hardware and software design, as well as failure analysis. Throughout this paper, we will discuss these observations and analyses within the context of a USB host controller on an embedded system.

Our research contributions are:

- A method for instrumenting device drivers to monitor peripheral operation
- A method for combining and relating the observed states of peripheral operation
- A classifier capable of estimating whether or not states are caused by ESD

The rest of the paper is as follows:

- A description of the software concept.
- An overview of hardware testing and the measurement process.
- A detailed explanation of the procedure for the software-hardware data logging.
- The Matlab tool capabilities.
- The findings of our work and what it means.
- Log classification and state prediction.

II. Related Work

Hardware ESD fault injection with direct injection and field injection probes is performed in [1] and [2]. These studies characterize integrated circuit (IC) immunity to ESD. The sensitivity threshold for each IC was determined by injecting ESD at increasing voltages and observing when errors occurred. In these studies, only user-visible errors, such as screen glitches or hardware resets, were studied.

Muchaidze et. al. [3] and Wang et. al. [4] extend this fault injection process by mapping the ESD sensitivity of the board. The injection probes are attached to a 3-D scanner that can

sweep them across the board. ESD is injected at each point on a grid and the device's susceptibility to ESD is noted. The resulting map can be used to identify traces and components that are at risk for ESD damage.

Sinha et. al. take a static approach to this problem in [5]. They develop a tool capable of analyzing circuit designs for ESD susceptibility that guides the optimal placement of sensitive components and traces. Unfortunately, static approaches such as this become prohibitively complex on modern circuit layouts.

While not directly related to ESD events, software-based and combined hardware and software system monitoring has been studied extensively. Watterson and Heffernan [6] outline research related to monitoring to perform runtime verification. The system state is monitored by some combination of hardware and software; this information is then used to verify that the system is operating within specification. A software-specific study of fault monitoring is presented in [7]. Delgado et. al. develop a taxonomy of runtime monitoring and discuss various system requirements for different monitoring techniques. Schroeder provides a tutorial in designing and implementing on-line monitoring systems in [8].

SIPE [9] laid the foundations for software monitoring of computer systems. It instrumented an event-driven multitasking system, recording the performance of the scheduler, the CPU, and various peripherals. Rota et. al. develop a software-based monitoring system for ATMs in [7]. They instrument each piece of hardware with kernel drivers that measure hardware state and performance. The data from these sensors are filtered and a runtime checker uses the filtered data to determine if the system is operating correctly. If not, recovery actions can be taken to restore sytem availability. Slye and Elnozahy discuss augmenting log-based rollback-recovery to support software interrupts in [10]. Log-based recovery is based on the software operating in a completely deterministic fashion based on what messages it receives. These messages are logged, so if the system fails, it can resume from the last message logged without any state inconsistencies. To expand their model to asyncronous operations and multi-threaded software, the authors instrument a threading library to log additional data, recording the nondeterminism caused by thread-based operations. They analyze the performance impact of this logging and verify that it performs as designed. Pont and Ong present seven patterns for improving embedded system reliability with watchdog timers [11]. A watchdog timer is a hardware timer that restarts the system if the timer is not reset within a specific time. Depending on what recovery behavior is desired and what sort of programs are run on the system, different approaches to 'feeding' the watchdog are taken.

Choudhuri and Givargis develop a mixed hardware and software approach for logging nondeterministic behavior in embedded systems [12]. They modify the compiler for an embedded system to log messages to an attached system that stores those messages in a flash storage chip. This combined approach greatly reduces processing overhead on the monitored system, making it applicable to very small embedded hardware. Reinbacher et. al. have created a tool that takes a software specification for an embedded system and produces both executables that perform that specification and a configuration for a hardware monitor [13]. The hardware monitor interfaces with the embedded CPU and its communication busses and verifies the operation of the system. Liu and Parthasarathi apply this concept to an entire cluster of machines [14]. Their hardware monitors a shared communication bus and records events. It can be used for debugging, monitoring, and fault recovery.

Yuan et. al. take a software approach to detecting ESD interference in microcontrollers [15]. They monitor the behavior of a phase-lock loop (PLL) embedded in the microcontroller; if the PLL unlocks, it can be assumed that the system has experienced an ESD shock. Their approach requires continuous polling of the PLL's status, adding a constant software overhead to the system. While it provides an excellent measure of ESD events on the microcontroller, it cannot measure peripheral ESD events because most peripherals do not containg a separate PLL that can be monitored by the microcontroller.

III. Our Approach

Inducing ESD events on an embedded system peripheral causes bits to flip in its data or control lines. These flipped bits can lead to changes in register values, loss of synchronization between peripherals and the CPU, or data corruption. All of these effects are visible to software running on the embedded system and thus should be detectable by monitoring software.

Our work focuses on monitoring these effects with software that allows normal system operation along with monitoring. We primarily study changes in register values, as those values control the behavior of the peripheral device. In effect, the tuple consisting of the values of each peripheral register represents a specific system state. Changes in register values represent changes in the peripheral's operating state. Some of these changes will be part of normal operation. When ESD is induced, however, we should see new abnormal states or unexpected transitions between normal states. These abnormal states and transitions can indicate that the system is experiencing ESD.

IV. Hardware Method

The board used during the testing is the FriendlyArm Mini2440 embedded development board with the Samsung S3C2440 ARM9 processor [16]. The USB host interface conforms to the Open Host Controller Interface specifications [17]. A modified Linux kernel, based on version 2.6.29 from the FriendlyArm website [16], was used. We set up the system with our logging software and connect it to a PC to control it during the tests. During testing, a standard flash drive holding a large file was connected to the board's USB port. To ensure that the host controller is active during ESD injection, we copied that file to or from the flash drive during tests.

ESD interference was injected using electric (E) field and magnetic (H) field probes powered by a transmission line pulse (TLP) generator. The E field does not have an orientation; we positioned it across the USB port or over the host controller IC. Because the magnetic fields generated by the H field probe

are polarized, we conducted tests with the probe in parallel with and perpendicular to the data and control lines. For each probe,multiple tests were run with varying pulse voltages. In addition, different size probes were used to adjust the intensity of the fields injected.

V. Software Monitoring

A. Initial Design

Our first design focused on directly reading USB register values from their physical memory addresses. We adapted and modified the Myregrw [18] software to better suit our needs as a softprobe for ESD. This software consists of a Linux kernel module and a program that communicates with it. The kernel module reads the values of requested physical memory addresses. The user-level program reads a configuration file specifying which addresses to request, repeatedly requests the data at those addresses, and stores that data to a file.

We configured Myregrw to record all the values between the addresses 0x49000000 and 0x49000014. The data in this window are the control and status registers for the USB host interface. We injected ESD into the host controller while Myregrw continuously sampled the registers. In theory, ESD-caused changes should appear in the recorded register values.

However, the sampling rate of this softprobe was not sufficient to gather ESD induced errors. Similarly to some of the works mentioned by Callanan in [19], bottleneck issues prevented this methodology from performing reasonably. We empirically determined that the sampling rate of the software is, on average, 342 times per second. We can assume the system executes one instruction per cycle and in the worse case would reset a register value after one instruction. The Mini2440 has a 400MHz clock. Thus $\frac{342}{400*10^6} * 100 = 0.000856\%$ would be the worst case probability of observing an error in the status register before it was changed. Considering this low probability and the lack of information recorded from our experiments, we devised a new measurement methodology with a higher sampling rate capable of recording additional register values.

A confounding issue with this approach is the competition for access to these values between the Myregrw driver and the USB host controller driver. By default, Linux drivers have execution priority over any user applications, meaning that it would be nearly impossible to read all the register values after an error, but before the USB host controller driver modifies the registers. Therefore, in addition to needing a faster sampling rate, the new methodology must account for Linux drivers modifying the control and status register values.

B. Improved Design

In the improved approach, we modified the drivers for the USB host controller to enable logging of relevant data. 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.

We first enabled the debugging configuration already present in the driver. In addition, we configured the driver to print the register values along with the name of the function to the system log before the execution of each function in the driver. These modifications allow us to observe not only register state changes but also the order in which different driver functions are called. We consider this to be minimally invasive to the driver software since it only requires trivial modifications to the driver code and does not affect the logic of the driver itself.

VI. Registers of Interest

Certain registers on the host controller give clear indications of ESD. In particular, we consider the values of the interrupt status, interrupt disable, and port status registers. The interrupt status register contains a few errors signals that can be seen in Figure I. There are three different types of error recorded here that can assist with identifying potential ESD events. The interrupt disable register shows how the hardware reacts to frame errors and overflows. The frame errors may be caused by flipped bits on the bus that are induced by ESD injection. The port status register indicates different types of detected errors. We see that the system will attempt to ignore some errors and try to recover by continuing to transmit packets. During normal operation there are only seven variations to the values recorded in the port status register, while we observed fourteen different values recorded during ESD injection, demonstrating that we are successfully recording errors while ESD interference is injected. Investigating the meanings of the register values indicated that the system was faulting from different errors, including frame overloading, connection loss, corrupt data, and connection status changing.

TABLE I. HcInterruptStatus Register Values Recorded

Value	Meaning	Comments
0x00000001	SchedulingOverrun	Possibly due to ESD bit flip
0x00000004	Start of a frame	Indicates normal state
0x00000006	Update of Information	Indicates normal state
0x00000010	UnrecoverableError	System has experienced an unexpected error
0x00000020	Unrecoverable Error Occurred	Need to investigate other registers for more info
0x00000024	Starting new state and error	Possibly ESD induced
0x00000030	RootHubStatusChange	Status change may be caused by ESD
0x00000060	Frame overflow and unrecoverable error	Good indicator of system fault
0x00000064	Attempting to start new frame despite errors	System trying to recover from ESD
0x00000066	Attempting to update registers with errors	Another sign of potential recovery

TABLE II. HcInterruptDisable Register Values Recorded

Value	Meaning	Comments
0x8000005A	Frame overflow, Unrecoverable Error	Clears bits
0x8000005E	Frame overflow, Clears enable register	Clears registers, sets start flags
0x8000001A	Start of new frame	Normal operation
0x8000001E	Starting new frame	Normal operation

TABLE III. HCRHPORTSTATUS0 REGISTER VALUES RECORDED

Value	Meaning	Comments
0x00000103	PortPowerStatus, PortEnableStatus, CurrentConnectedStatus	Normal operation
0x00000101	PortEnableStatus is disabled	Need to investigate other registers for more info
0x00000100	Port has power no devices connected	Shows that connection was disturbed
0x00030100	Turning power on to all ports connection status changed	Possibly ESD induced
0x00030101	Frame overflow and unrecoverable error	Good indicator of ESD
0x00100103	Attempting to update registers with errors	Sign of potential recovery
0x00010100	Attempting to start new frame despite errors	Sign of potential recovery
0x00020101	Change in portEnableStatus	Indicates normal state
0x00010101	Connect or disconnect occured	Only occurred in ESD-exposed logs
0x00000111	Data set in reserved space	Sign that ESD injection flipped bits
0x00000113	Data set in reserved space	Another sign of ESD injection flipping data bits
0x00020103	Indicates connection reset	Only occurred under ESD exposure, possibly ESD induced
0x00120101	PortResetStatusChange	The port connection reset command was sent

VII. ANALYSIS

Once we performed a series of tests, we copied the resulting log files off of the test board for further analysis. These log files consist of lines each having a timestamp, register name, and associated register value. We must first convert these log files into lists of system states defined by the set of values of all recorded registers at a specific time. Because a state is created each time the register values are recorded, each log will contain repeated states when the driver repeats an operation. We will need to find these equivalent states and combine them to derive a state transition graph. In addition, since certain register values change every time the system is restarted, we will perform deduplication to compare execution traces from different tests. The analysis code's operation can be summarized as follows:

- Parse the log file to create states based on the registers' information.
- Deduplicate these execution traces to derive a per-test execution graph.
- Deduplicate the execution graphs of different tests to derive a universal execution graph.
- Using this execution graph and each tests' execution trace, perform statistical analysis on the testing data.

A. Individual Log Files

The first stage in the analysis involves parsing a log file for one simulation. We read the register values for each state from the log file. After creating all the states, we use the list of states to create a list of unique states and record the order in which these states were reached. States that were executed sequentially show transitions between unique states; combined with the unique state graph, they form a state machine for the host controller's operation. For statistical analysis, we also record the number of times each transition is taken.

B. Globally Universal States

The next element of analysis is combining the data from each test run into a global state machine. The process is very similar to that for developing the unique state graph for each log. Certain registers for the host controller contain memory addresses that change every time the driver is reloaded. These registers are HcPeriodCurrentED, HcBulkCurrentED, HcFmRemaining, HcHCCA, HcControlHeadED, HcControlCurrentED, HcBulkHeadED, HcFmNumber, and HcDoneHead Since these values will differ from test to test, they should not be considered when comparing states from different logs. However, changes in these values in a single test may indicate ESD and therefore result in a new globally unique state being created.

C. Graph Analysis

We divide the data collected from test runs into two groups: baseline and ESD-exposed. Base logs are logs of the system operating normally and provide us with the system's expected state machine. ESD-exposed logs then show how the system transitions into and out of erroneous behavior due to ESD exposure.

After the creation of the list of globally unique states, we can analyze the ESD-exposed and baseline logs individually to observe how system behavior differs between them. To gain an understanding on how the system varies between normal operation and operation with ESD interference, two universal lists are created; one from baseline logs and one from ESD-exposed logs. Thes two sets of states are subtracted to determine which states are only observed during ESD injection. These state sets can be used to show where during operation the system transitioned into a state that is very likely to have been caused by ESD.

VIII. RESULTS

Figure 1 shows sample baseline and ESD-exposed state graphs. The set of nodes and solid arcs on the left of the figure is the state graph of the baseline log. The right of the figure consists of the additional states and transitions present in a sample ESD-exposed log. Green states and solid edges indicate states and transitions present in the baseline logs. Red states and dashed edges indicate states and transitions appearing only in ESD-exposed logs. This state graph demonstrates several effects of ESD on the system state: transitions to abnormal states, transitions between abnormal states, abnormal transitions between normal states, and transitions from abnormal to normal states.

Consider how we should expect the system to behave under normal conditions and under ESD exposure. Normally, it should have a small number of common code paths and some edge case handling. Under ESD exposure, we should see a large number of different anomalous states caused by different bits being flipped. Figure 2 shows the average number of ocurrences per log file of states from normal-condition logs and ESD-exposed logs. We see a few normal states that are very common, with a small tail of less common states. There are far

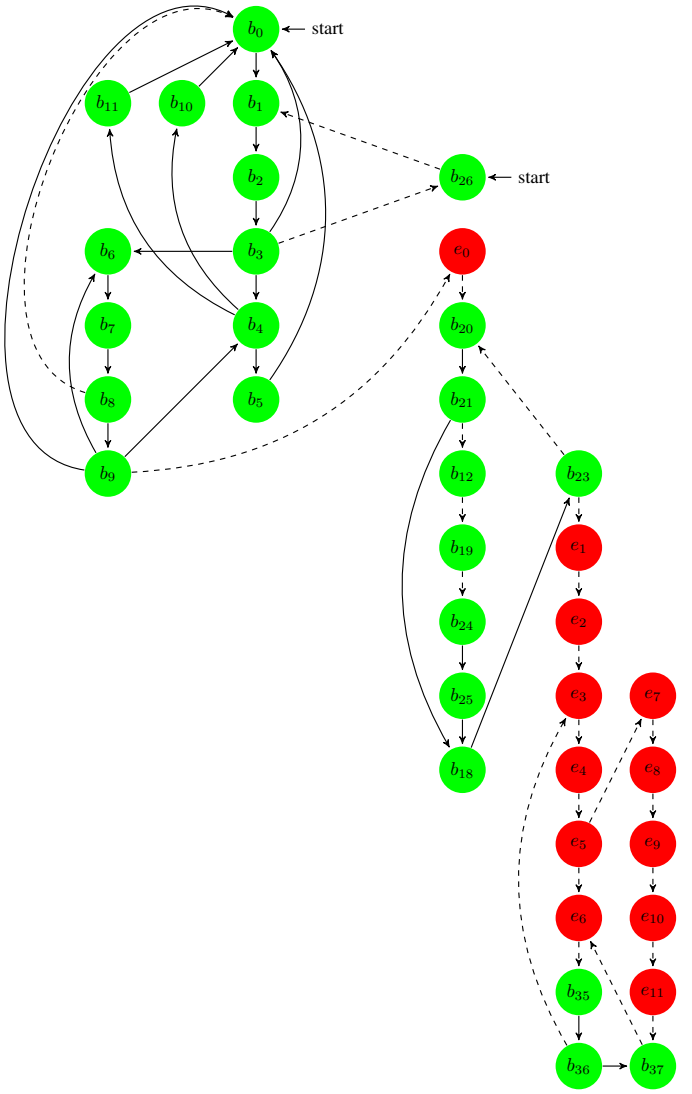

Fig. 1. State execution graph of one baseline log and one ESD-exposed log

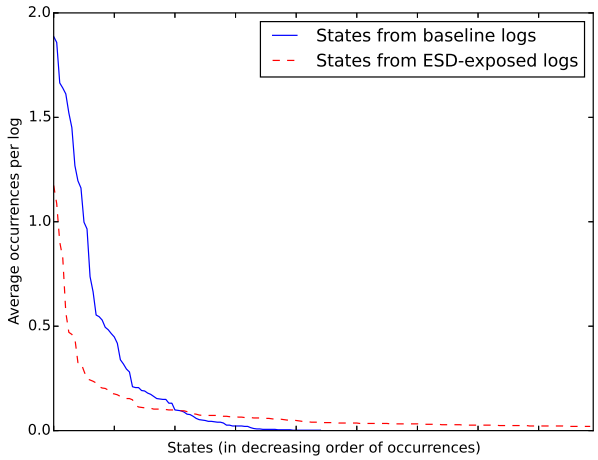

Fig. 2. Average State Occurrences Per Log

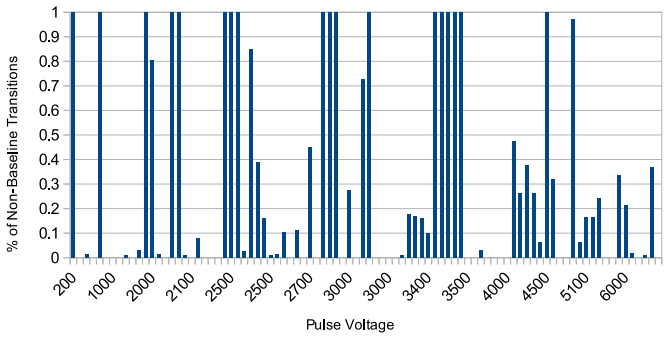

Fig. 3. Relationship between pulse voltage and ESD-caused transitions

more unique states in ESD-exposed logs, and they are far less likely to occur. (We have omitted half of the ESD-exposed state tail to make the interesting portion of the graph more legible.) This graph provides a sanity check for our methodology; we can see that the data we have collected is behaving as we expect it to.

Figure 3 compares the TLP pulse voltage with the percentage of transitions to or from states not in the baseline logs. The lack of a clear relationship between observed ESD coupling and pulse voltage indicates that there are confounding factors between ESD exposure and system behavior. These factors may include field type and orientation, injection location, pulse frequency, and the operation being performed by the host controller at the time of injection. As well, the host controller software may recover from some ESD-induced faults on its own. Finally, the ESD injection may cause the system to crash almost instantaneously, so the resulting state log will have relatively few states caused by ESD. More work is needed to assess the effect each of these factors has on system operation.

A. *Electric Field Injection*

The E-field EZ 3 probe was used to inject interference on the USB connection. This probe was connected to a TLP that generated interference pulses every 0.25 seconds. The pulse voltage was adjusted in 500 volt increments per test.

Various errors were located in the following registers: operating mode register, interrupt status, interrupt enable, interrupt disable, and port status 0. These registers indicate that both overflow errors and unrecoverable errors occurred. The occurrence of these errors indicates the system crashed as a result of data overflow. Several of these errors can be seen in Table IV.

B. *Magnetic Field Injection*

Magnetic field injection requires the use of a different probe. The rest of the system configuration is the same as for the E-

TABLE IV. ERROR STATE VALUES RECORDED

Register	Error State	Error State	Error State	Error State
HcControl	0x93	0x83	0x83	0xa3
HcCommandStatus	0x0	0x0	0x0	0x0
HcInterruptStatus	0x60	0x24	0x64	0x20
HcInterruptEnable	0x8000001a	0x8000005a	0x8000001e	0x8000005a
HcInterruptDisable	0x8000001a	0x8000005a	0x8000001e	0x8000005a
HcFmInterval	0xa7782edf	0xa7782edf	0xa7782edf	0xa7782edf
HcPeriodicStart	0x2a2f	0x2a2f	0x2a2f	0x2a2f
HcLSThreshold	0x628	0x628	0x628	0x628
HcRhDescriptorA	0x2001202	0x2001202	0x2001202	0x2001202
HcRhDescriptorB	0x0	0x0	0x0	0x0
HcRhStatus	0x8000	0x8000	0x8000	0x8000
HcRhPortStatus0	0x103	0x111	0x113	0x103
HcRhPortStatus1	0x100	0x100	0x100	0x100

field injection.

The H Field HX 5 probe was used first. The pulse voltage level needed to be increased to generate errors similar to those experienced with the E field probe. During E field injection tests kernel panics could be observed at as low as 3kV, whereas for H field injection the lowest voltage needed to experience a kernel panic was 6kV. An analysis of the results confirms that the system was able to operate normally with smaller injections of interference. We summarize the effect of voltage on observed errors in Table V. Injecting the fields both in parallel and perpendicular generated the same results.

TABLE V. H FIELD RESULTS

Probe	Voltage	Observed Result
H Field High ESD HX 5	1000	No errors observed.
H Field High ESD HX 5	2000	No errors observed.
H Field High ESD HX 5	3000	No errors observed.
H Field High ESD HX 5	4000	USB Read errors.
H Field High ESD HX 5	5000	EMI messages from driver.
H Field High ESD HX 5	6000	Kernel Panic.

IX. CLASSIFIER

So far, we have focused on determining how the system behaves when exposed to ESD. However, we may want to reverse this relationship and determine whether or not the system is being exposed to ESD based on its behavior. We present a preliminary finite state classifier for the host controller state data. While the approach here is limited to offline training and classification, it has been designed with the ability to be easily converted to real-time operation. As such, the classification process is computationally inexpensive in order to reduce overhead on an embedded system.

The finite state classifier itself has two states: 'normal' and 'ESD-exposed', representing the status of the system over some time period. As the host controller operates, each state the controller enters causes a transition between classifier states. The classifier has a concept of 'certainty' that describes the likelyhood that the current classifier state is correct (e.g. the likelyhood that the system is experiencing ESD when the classifier is in the 'ESD-exposed' state). Certainty is quantified as a value between -1 and 1, with -1 representing complete certainty that the classification is 'normal' and 1 representing complete certainty that the classification is 'ESD-exposed'. An ESD event would thus appear as an increasingly positive certainty that the system is ESD-exposed. Recovery from such an event would appear as an increasingly negative certainty.

A. Training

In order to deduce this from our data, we must consider three classes of states: states appearing only in baseline logs, states only appearing in ESD-exposed logs, and states appearing in both. We say we are certain that states only in baseline logs indicate normal behavior, and states only in ESD-exposed logs are a certain indicator of ESD exposure. Certainty is implemented by assigning each system state a weight. Weights are selected to be within the range $[-1, 1]$ such that a positive weight value indicates an ESD-exposed state and a negative weight value indicates a normal state.

Weight value assignment must compensate for the fact that the training data will contain more logs from error operation than base operation, since we performed more ESD-exposed tests than baseline tests. Weights are assigned as follows:

Let B and E be the sets of all base and error states, respectively.

Define constants for base states $w_b = -|E|$ and for ESD-exposed states $w_e = |B|$.

For a unique state i, let b_i be the number of times state i is present in B, and e_i be the number of times it is present in E.

For each unique state i, the normalized weight is defined as:

$$w_i = \frac{w_b * b_i + w_e * e_i}{|w_b * b_i| + |w_e * e_i|}$$

which can also be written as

$$w_i = \frac{-|E| * b_i + |B| * e_i}{|E| * b_i + |B| * e_i}$$

B. Classification

We can use these weights to build a classifier for the logs we have collected. Currently, our time window for classification is the length of the log file being tested, but it could be easily changed to a moving-window classifier for real-time applications.

The weights can be used to classify a log as follows:

Let S_L be the set of unique states in log L, and s_i the number of times state i appears in the log.

The log's classification, C_L, is then

$$C_L = \frac{\sum_i^{S_L} w_i * s_i}{|L|}$$

If C_L is positive, we classify the log as having experienced ESD, and if it is negative, we classify it as having not experienced ESD.

Because the training data for the classifier is not guaranteed to contain all states, certain states in the test data may not have assigned weights. We default these weights to 0, so they have no effect on the certainty of the classifier.

Classifier precision can be measured by the percentage of states for which weights are defined. We define accuracy by whether baseline logs are classified as normal operation and ESD-exposed logs are classified as ESD-exposed operation.

We tested this procedure over 22 base logs and 113 ESD-exposed logs. Classifier performance over k-fold verification

with $k = 10$ was an 66.1% accuracy and 82.8% minimum precision.

C. System State Transitions

We hypothesize that the behavior of the host controller is non-Markov; that is, knowing how the system reached the current state may allow us to predict the states it will transition to with higher accuracy. In addition, ESD exposure may cause the system to abnormally transition between two normal states. This effect of ESD is not captured in the current classifier design. We can provide some contextual knowledge for the classifier by classifying system transitions instead of system states. Additionally, we can extend the concept of a transiton, which is a 2-tuple of states, to an 'n-sition', an n-tuple of states in the order they were executed. In graph theory terms, an 'n-sition' is a path of length $n - 1$. One disadvantage of adding context in this fashion is the increasing likelihood of encountering an n-sition in the test data that was not present in the training data. Thus, we must choose a tradeoff between increasing accuracy and decreasing precision. Another disadvantage is the increased storage overhead for the weight table, which must be able to fit into memory on an embedded system. Table VI shows classifier performance for different n-sitions, again averaged over $k = 10$ folds. The $n = 1$ data is the same as that presented in the previous section.

TABLE VI. CLASSIFIER PERFORMANCE FOR DIFFERENT n-SITIONS

n	Avg. Accuracy	Avg. Precision
1	66.1%	82.8%
2	84.6%	63.2%
3	85.3%	42.9%
4	86.1%	26.6%
5	87.4%	17.7%
6	87.2%	12.9%

D. Delta

During testing, we determined that the classifier was performing poorly in part due to an underestimation of the weights of states indicating ESD exposure. We introduced the δ parameter as a means of adjusting the weight distribution.

Base and ESD-exposed constants are implemented as $w_b = -|E| - (|E| + |B|) * \delta$ and $w_e = |B| + (|E| + |B|) * \delta)$.

Figure 4 shows the change in the distribution of weights without δ and with $\delta = 0.557$.

Running $k = 10$-fold cross-validation for several n-sitions and values of δ between 0 and 1, the classifier performs as shown in Table VII.

TABLE VII. CLASSIFIER PERFORMANCE FOR DIFFERENT n-SITIONS WITH δ

n	Avg. Accuracy	Avg. Precision	Best δ
1	86.1%	82.8%	0.467
2	91.5%	63.2%	0.557
3	90.0%	42.9%	0.138
4	90.7%	26.6%	0.138
5	90.4%	17.7%	0.125
6	90.3%	12.9%	0.25

As we expect, δ has no effect on precision, but it does have a large effect on accuracy. It allows us to reduce the amount of context the classifier needs to perform well, allowing us to have a more precise classifier without losing accuracy.

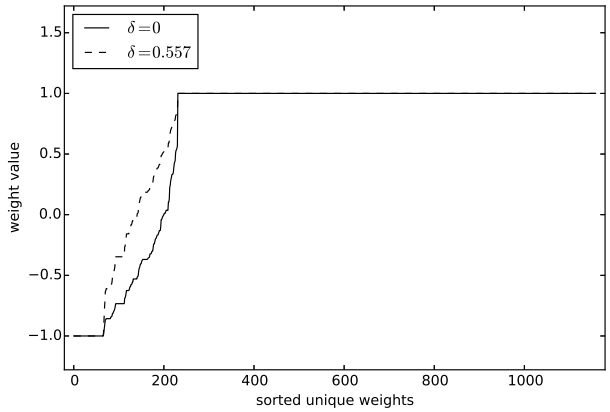

Fig. 4. Weights With and Without δ

X. CONCLUSION

We have presented a software-based methodology for detecting ESD events on embedded system peripherals. This methodology approximates the state of the peripheral by reading the registers it exposes to the CPU with an instrumented kernel driver for the peripheral.

We applied this methodology to a USB Host Controller on an embedded system running Linux. We demonstrated that we are able to observe states and transitions that the system experiences only when exposed to ESD.

The relationship between the recorded errors and ESD can be reversed. Doing so allows us to predict, based on the errors that the software experiences, when and where the system experiences ESD. We can apply this in several ways: 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). In addition, software can be written to recover from error states in a more efficient and automatic fashion. Software may also be able to compensate for the effects of ESD, allowing operation to continue in adverse environments, although at the cost of reduced performance and more software overhead.

Our work contains a preliminary classifier that can be used to identify this relationship between system errors and ESD exposure. It is especially designed to provide a foundation for ESD-robust software to be developed on. The next step beyond this research would be to implement a real-time classifier running on the embedded system.

Other avenues for research include correlating particular error states with ESD injection on a specific location on the board and applying this methodology to other peripherals and embedded systems.

REFERENCES

[1] Z. Li, J. Xiao, B. Seol, J. Lee, and D. Pommerenke, "Measurement methodology for establishing an ic esd sensitivity database," in *Electromagnetic Compatibility (APEMC), 2010 Asia-Pacific Symposium on*, pp. 1051–1054, april 2010.

[2] D. P. J. Koo and G. Muchaidze, "Finding the root cause of an esd upset event," 2006.

[3] 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, pp. 268–276, May 2008.

[4] 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*, vol. 2, pp. 350–355, August 2005.

[5] S. Siha, H. Swaminathan, G. Kadamati, and C. Duvvury, "An automated tool for detecting esd design errors," in *Electrical Overstress/Electrostatic Discharge Symposium Proceedings*, p. 208 217, October 1998.

[6] C. Watterson and D. Heffernan, "Runtime verification and monitoring of embedded systems," *Software, IET*, vol. 1, no. 5, pp. 172–179, 2007.

[7] N. Delgado, A. Gates, and S. Roach, "A taxonomy and catalog of runtime software-fault monitoring tools," *IEEE Transactions on Software Engineering*, vol. 30, no. 12, pp. 859–872, 2004.

[8] B. Schroeder, "On-line monitoring: a tutorial," *Computer*, vol. 28, p. 72 78, June 1995.

[9] W. R. Deniston, "Sipe: A tss/360 software measurement technique," in *Proceedings of the 1969 24th national conference*, ACM '69, (New York, NY, USA), pp. 229–245, ACM, 1969.

[10] J. Slye and E. Elnozahy, "Support for software interrupts in log-based rollback-recovery," *Computers, IEEE Transactions on*, vol. 47, no. 10, pp. 1113–1123, 1998.

[11] M. J. Pont and H. Ong, "Using watchdog timers to improve the reliability of ttcs embedded systems: Seven new patterns and a case study," in *Proceedings of VikingPLOP*, 2002.

[12] S. Choudhuri and T. Givargis, "Flashbox: a system for logging nondeterministic events in deployed embedded systems.," in *SAC* (S. Y. Shin and S. Ossowski, eds.), pp. 1676–1682, ACM, 2009.

[13] T. Reinbacher, M. Horauer, and A. Steininger, "A runtime verification unit for microcontrollers," in *System, Software SoC and Silicon Debug Conference (S4D)*, p. 1 6, September 2012.

[14] A.-C. Liu and R. Parthasarathi, "Hardware monitoring of a multiprocessor system," *IEEE Micro*, vol. 9, no. 5, p. 44 51, 1989.

[15] 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.

[16] "Downloads - friendlyarm." Accessed: 27/02/2013.

[17] "1394 Open Host Controller Interface Specification - Microsoft." http://www.microsoft.com/whdc/system/bus/1394/OHCI.mspx. Accessed: 2015-01-23.

[18] "edwinrong/myregrw." https://github.com/edwinrong/myregrw. Accessed: 2015-01-23.

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