A Reverse-Boot Enabled Embedded Computing System on SoC-FPGA for Prognostics and Health Management: An Application on Li-ion Batteries

Ruolin Zhou
Department of Electrical and Computer Engineering
University of Massachusetts
Dartmouth, MA, USA
ruolin.zhou@umassd.edu

Yuan Liu
Department of Electrical and Computer Engineering
Western New England University
Springfield, MA, USA

Eric Brodeur
Department of Electrical and Computer Engineering
Western New England University
Springfield, MA, USA

Zhaojun Li
Department of Industrial Engineering and Engineering Management
Western New England University
Springfield, MA, USA
zhaojun.li@wne.edu

Jian Guo
Department of Industrial Engineering and Engineering Management
Western New England University
Springfield, MA, USA
zhaojun.li@wne.edu

***Abstract*—In this paper, a secure embedded computing system (ECS) on System-on-Chip Field Programmable Gate Array (SoC-FPGA) through controlling boot sequence is designed. An application of prognostics and managing health conditions of Li-ion batteries has been implemented on the SoC-FPGA based ECS. Specifically, to increase the security of the ECS, different booting sequences have been implemented and evaluated. A prognostic algorithm has been implemented on the SoC-FPGA based ECS platform to diagnose the health condition of Lithium-ion batteries and predict the remaining useful life of these batteries. The proposed secure ECS is scalable to predict the health condition of larger scale engineering systems to optimally manage and maintain the engineering systems before systems fail.**

***Index Terms*—SoC-FPGA, security, boot sequence, embedded computing system, prognostics and health management**

I. Introduction

Embedded computing systems (ECS), which mainly contains hardware, application software, and real-time operating system, are usually designed for some specific tasks with real-time performance constrains, rather than being a general-purpose computer for multiple tasks. The market for ECS grows at about 14% rate worldwide since the embedded computing systems will be responsible for 90% of all automation innovations, such as autonomous vehicles, automated medical equipment, avionics systems, smart home appliances, military systems, and so on [1]. Over 6 billion new microprocessors are used each year whereas less than 2% (about 100 million per year) of the microprocessors are used in general-purpose computers and over 98% are for embedded systems for computation and control purposes [2].

In addition, to perform computation efficiently and effectively on ECS platform, parallel processing, where a large problem can be solved faster by breaking it down to smaller pieces and solving the smaller pieces simultaneously, is outperformed [3]. Such a parallel processing can be implemented through GPU (Graphics Processing Unit) or ASIC (Application Specific Integrated Circuit). However, GPU consumes much higher power and has unpredictable latency [4] [5]. ASIC may meet design requirements in terms of latency and SWaP (Size, Weight, and Power) constraints, but it is a huge, costly, and risky effort since it is not a reconfigurable and reprogrammable chip. One chip is fabricated for just one application. Therefore, Field Programmable Gate Array (FPGA), which has a high capacity for parallelization and pipeline processes, is a better hardware solution. Meanwhile, FPGA can also guarantee low time from development to market introduction with high flexibility and performance [6] [7], and can be used as peripherals to microprocessors to carry out specific processes that a microprocessor has troubles to handle efficiently.

Therefore, System-on-Chip FPGA (SoC-FPGA) is employed as the embedded computing platform in this paper. To increase the security of the ECS, different booting sequences have been implemented and evaluated. A prognostic algorithm has been implemented on the SoC-FPGA ECS platform to diagnose the health condition of Lithium-ion batteries and predict the remaining useful life of these batteries. The proposed secure ECS is scalable to predict the health condition of larger scale engineering systems to optimally manage and maintain the engineering systems before systems fail.

The rest of the paper is organized as follow: Section II describes different boot sequences on SoC-FPGA; Section III illustrates experiment results of booting from SD card and reverse boot, booting from FPGA; Section IV demonstrates the prognostics on the SoC-FPGA ECS system; and Section V concludes the paper.

II. Overview of the Boot Sequence on SoC-FPGA

Intel DE1-SoC FPGA development kit is studied because it supports reverse boot, booting from FPGA. The booting sequence of an embedded system on Intel DE1-SoC FPGA usually consists of four stages: BootROM, Preloader, U-Boot, and Linux. Specifically, BootROM is used to detect the boot source and initialize all required hardware components in order to boot up the Preloader which is the next stage boot software. Preloader is part of U-Boot that is an open source and primary boot loader used in embedded systems. Preloader is responsible for further hardware initialization, including synchronous dynamic random-access memory (SD-RAM). It

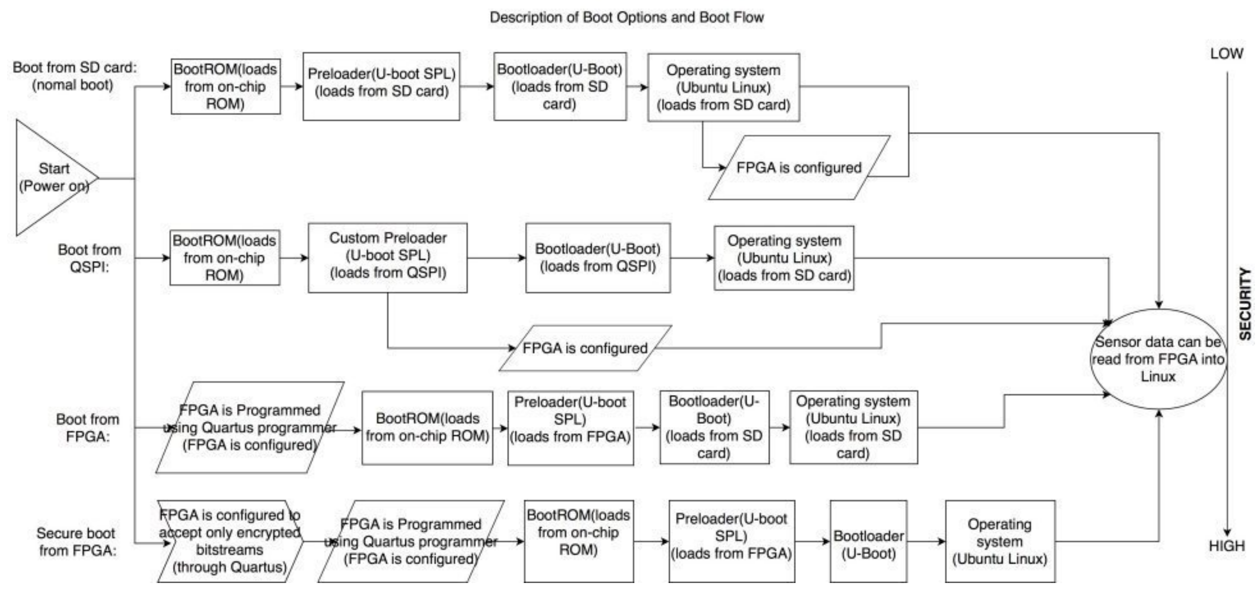

Description of Boot Options and Boot Flow
Boot from SD card: (nomal boot)
BootROM(loads from on-chip ROM)
Preloader(U-boot SPL) (loads from SD card)
Bootloader(U-Boot) (loads from SD card)
Operating system (Ubuntu Linux) (loads from SD card)
FPGA is configured
Start (Power on)
Boot from QSPI:
BootROM(loads from on-chip ROM)
Custom Preloader (U-boot SPL) (loads from QSPI)
Bootloader(U-Boot) (loads from QSPI)
Operating system (Ubuntu Linux) (loads from SD card)
FPGA is configured
Sensor data can be read from FPGA into Linux
Boot from FPGA:
FPGA is Programmed using Quartus programmer (FPGA is configured)
BootROM(loads from on-chip ROM)
Preloader(U-boot SPL) (loads from FPGA)
Bootloader(U-Boot) (loads from SD card)
Operating system (Ubuntu Linux) (loads from SD card)
Secure boot from FPGA:
FPGA is configured to accept only encrypted bitstreams (through Quartus)
FPGA is Programmed using Quartus programmer (FPGA is configured)
BootROM(loads from on-chip ROM)
Preloader(U-boot SPL) (loads from FPGA)
Bootloader (U-Boot)
Operating system (Ubuntu Linux)
LOW
SECURITY
HIGH

Fig. 1. Different Choices of Configuration Modes using Intel DE1-SoC-FPGA

resides in writable memory, and can be modified via synthesis tool if so desired. Preloader can access various well tested drivers available for U-Boot. The next stage, U-Boot, shares some of the same responsibilities as the Preloader. However, the main functions of U-Boot are (1) setting up the operating system (OS) environment; (2) fetching the OS image from memory or Ethernet; (3) storing the boot image to SDRAM and passing control over to subsequent bootloader; and (4) providing a console for uses to modify boot arguments [8]. Different choices of configuration modes as well as sequences of booting the system have been summarized in Fig. 1.

III. Experiment Results of Regular Boot and Reverse Boot

A. Test Methodology

Power consumption for each boot method has been measured by first measuring the voltage drop across a shunt resistor wired in series with the board, and deriving the current through the board using Ohm's law: $I = V/R$, where $I$is the current through the load and the shunt, V is the voltage drop across the shunt, and R is the known resistance of the shunt.

Power consumed is then calculated using the electrical power equation (or Joule's First Law): $P = I \times V$, where P is the power consumed by the load, I is the current through the load, and V is the known voltage drop across the load (as measured by a lab multimeter). Fig. 2 shows a schematic of the test circuit.

B. Boot from SD Results

Booting from SD is a regular and commonly used boot sequence. It has been achieved with resources from the Altera DE1-SoC CD-ROM [9]. The testing methodology is followed by replacing the load with DE1-SoC.

Three tests have been conducted for booting from SD. Fig. 3 is the experiment results of test 2. The value of shunt resistor in test 2 is measured to be 1.02 Ω, and voltage from the power supply into the DE1-SoC is measured to be 12.199 Volts using a multimeter.

To determine the time it takes at each boot stage, UART (Universal Asynchronous Receiver/Transmitter) serial communication through the PuTTY serial console application is applied. From the information displayed through PuTTY, it is able to determine which boot stage the device is in at any given time. The time in each boot stage (power on to U-Boot completed, while loading Linux, and idle state post-booting Linux completely) is measured and summarized in Fig. 3. The top graph of Fig. 3 has been annotated to show the results of the boot stage timing test. It should be noted that there is a 5 second wait programmed into the Bootloader code that occurs before the operating system is loaded. This delay could be removed to improve boot time if so desired.

The average current draw and power consumption for the different boot stages are summarized at the top portion of Fig. 3. The results of boot from SD test 2 show that the

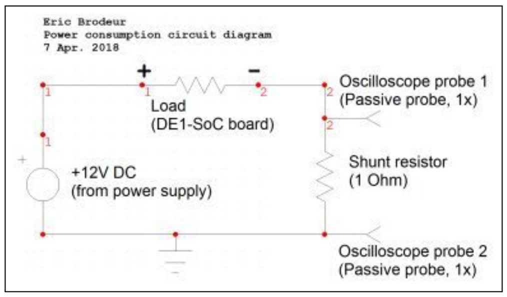
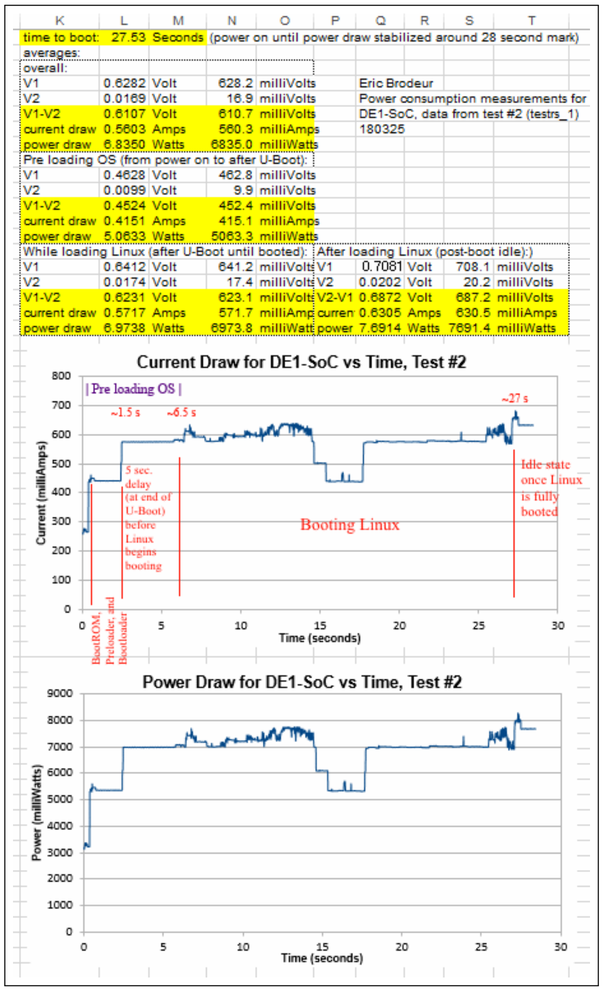

Eric Brodeur
Power consumption circuit diagram
7 Apr. 2018
+
-
Load (DE1-SoC board)
Oscilloscope probe 1 (Passive probe, 1x)
+12V DC (from power supply)
Shunt resistor (1 Ohm)
Oscilloscope probe 2 (Passive probe, 1x)

Fig. 2. Schematic of power consumption testing circuit made in ICAPS/4. Ground leads of both scope probes are connected to GND in the circuit.



K	L	M	N	O	P	Q	R	S	T
time to boot:	27.53	Seconds	(power on until power draw stabilized around 28 second mark)						
averages:									
overall:									
V1	0.6282	Volt	628.2	milliVolts		Eric Brodeur			
V2	0.0169	Volt	16.9	milliVolts		Power consumption measurements for			
V1-V2	0.6107	Volt	610.7	milliVolts		DE1-SoC, data from test #2 (testrs_1)			
current draw	0.5603	Amps	560.3	milliAmps		180325			
power draw	6.8350	Watts	6835.0	milliWatts					
Pre loading OS (from power on to after U-Boot):									
V1	0.4628	Volt	462.8	milliVolts					
V2	0.0099	Volt	9.9	milliVolts					
V1-V2	0.4524	Volt	452.4	milliVolts					
current draw	0.4151	Amps	415.1	milliAmps					
power draw	5.0633	Watts	5063.3	milliWatts					
While loading Linux (after U-Boot until booted):					After loading Linux (post-boot idle):)				
V1	0.6412	Volt	641.2	milliVolts	V1	0.7081	Volt	708.1	milliVolts
V2	0.0174	Volt	17.4	milliVolts	V2	0.0202	Volt	20.2	milliVolts
V1-V2	0.6231	Volt	623.1	milliVolts	V2-V1	0.6872	Volt	687.2	milliVolts
current draw	0.5717	Amps	571.7	milliAmp	curren	0.6305	Amps	630.5	milliAmps
power draw	6.9738	Watts	6973.8	milliWatt	power	7.6914	Watts	7691.4	milliWatts

Current Draw for DE1-SoC vs Time, Test #2
| Pre loading OS |
~1.5 s
~6.5 s
~27 s
5 sec. delay (at end of U-Boot) before Linux begins booting
Booting Linux
Idle state once Linux is fully booted
BootROM, Preloader, and Bootloader
Current (milliAmps)
Time (seconds)
Power Draw for DE1-SoC vs Time, Test #2
Power (milliWatts)
Time (seconds)

Fig. 3. Screenshot showing summary of data analysis for test 2.

board draws on average 5.1W of power while in the BootROM, Preloader, and Bootloader stages (collectively), 7.0W of power while booting Linux, and 7.7W of power after booting Linux completely and programming the FPGA. Most of the time in the boot sequence is spent booting Linux, and the highest power draw occurs after Linux has booted completely. The results of tests 1 and 3 were consistent with the results from test 2.

C. *Boot from FPGA Results*

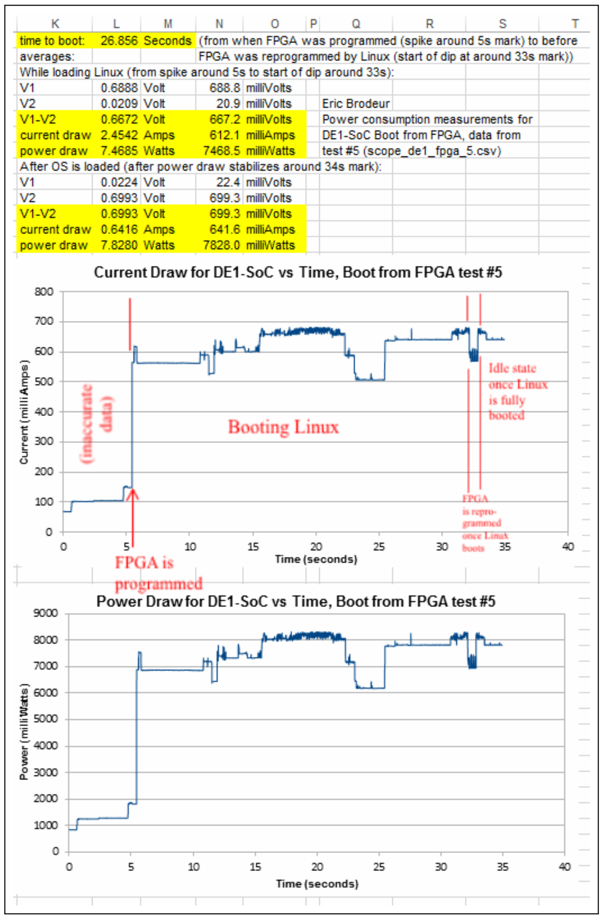
Reverse boot, booting from FPGA [9], is more secure than the regular boot up since the FPGA is programmed first and then the system boots up which forces the processor to load an application from a known location/address. Similar data has been collected when the board is configured to boot from FPGA. Seven sets of data are collected for booting from FPGA. The result of test 5 is shown in Fig. 4, where the value of shunt resistor is measured to be 1.02 Ohms, and voltage from the power supply into the DE1-SoC is measured to be 12.201 Volts using a multimeter. Analysis is done for when the Linux OS is booting as well as once Linux has booted fully and is idle. The top graph has been annotated to show the stages of the boot sequence. Notice that the measurements spike up near the 5 second mark; this is when the USB-Blaster II cable (for configuring FPGA) is removed from the board, and data before this spike is therefore not accurate.

A technical difficulty has been encountered while taking data of the boot from FPGA. Specifically, the process of programming the FPGA via Quartus programmer requires the use of a USB AB cable (recognized by Quartus as USB-Blaster II). The test circuit cannot accurately measure the current draw/power consumption when the USB-Blaster II cable is connected to the board. The differential voltage across the shunt resistor is read by the scopes to be almost one third of what the voltage should be. However, with the cable disconnected from the DE1-SoC, measurements are accurate. This issue is the reason why more data sets have been taken for booting from FPGA compared to booting from SD. A work-around for this issue is implemented by running several different types of tests:

- Datasets 2 and 7 measure the full boot sequence from power on to post-boot Linux with the USB cable connected to the DE1-SoC. For these datasets, the power consumption readings are inaccurate, but the relative changes in current draw/power consumption can be observed.
- Datasets 1, 4, and 5 measure the full boot sequence as well with the USB Blaster II cable disconnected immediately after programming the FPGA via Quartus. The power/current draw for these tests is inaccurate for the beginning portion of data when the cable is connected, but accurate otherwise.
- Datasets 3 and 6 measure only from the point when the FPGA has been programmed through Quartus until the Linux OS fully boots. The point at which it is determined that Linux has fully booted is when seven-segment displays on the board turned off briefly while the board is being reprogrammed from Linux. It is because a FPGA reconfiguration is part of a script that runs at the end of the Linux boot sequence for testing purpose. Tests 3 and 6 give accurate boot time and power consumption when booting Linux.

By observing the results of each type of test in relation to the others, an accurate view of power consumption and boot time when booting from FPGA is formed despite the USB cable issue.

Comparing the results of the boot from SD and boot from FPGA testing, power consumption and boot time are of minimal difference between booting from SD and booting from FPGA, with FPGA booting slightly faster and secure, and consuming slightly more power on average. However, it should be noted that the FPGA is programmed much earlier in the boot sequence when booting from FPGA, which could improve boot time if setup or calculations need to be performed on the FPGA device before the user program can execute, since in this case the FPGA calculations would be running in parallel with the processor boot sequence.



K	L	M	N	O	P	Q	R	S	T
time to boot:	26.856	Seconds	(from when FPGA was programmed (spike around 5s mark) to before						
averages:			FPGA was reprogrammed by Linux (start of dip at around 33s mark))						
While loading Linux (from spike around 5s to start of dip around 33s):									
V1	0.6888	Volt	688.8	milliVolts					
V2	0.0209	Volt	20.9	milliVolts		Eric Brodeur			
V1-V2	0.6672	Volt	667.2	milliVolts		Power consumption measurements for			
current draw	2.4542	Amps	612.1	milliAmps		DE1-SoC Boot from FPGA, data from			
power draw	7.4685	Watts	7468.5	milliWatts		test #5 (scope_de1_fpga_5.csv)			
After OS is loaded (after power draw stabilizes around 34s mark):									
V1	0.0224	Volt	22.4	milliVolts					
V2	0.6993	Volt	699.3	milliVolts					
V1-V2	0.6993	Volt	699.3	milliVolts					
current draw	0.6416	Amps	641.6	milliAmps					
power draw	7.8280	Watts	7828.0	milliWatts					

Current Draw for DE1-SoC vs Time, Boot from FPGA test #5
Current (milli Amps)
Time (seconds)
(inaccurate data)
Booting Linux
Idle state once Linux is fully booted
FPGA is reprogrammed once Linux boots
FPGA is programmed
Power Draw for DE1-SoC vs Time, Boot from FPGA test #5
Power (milliWatts)
Time (seconds)

Fig. 4. Screenshot showing summary of data analysis for boot from FPGA test 5.

IV. Algorithm Implementation on SoC-FPGA Based ECS

Based on our previous research in [10], we implement a prognostic algorithm on the SoC-FPGA based ECS to predict the remaining useful life of Li-ion batteries in this section. Fig. 5 shows our battery testing equipment, which consists of an Arbin battery test machine, environmental chamber, battery module and the SoC-FPGA based ECS. Part of collected charge/discharge battery data is shown in Table I.

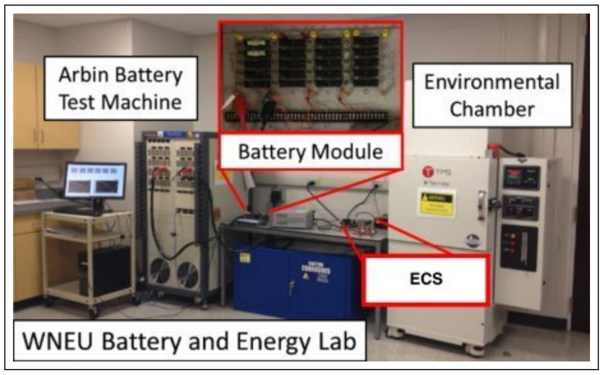


Fig. 5. Batteries Testing

TABLE I
Battery Experiment Results (part)

Battery Experiment Results						
term	cycle	capacity	battery	sqrtcycle	logcycle	interaction
1	1	1.162	33	1	0	0
2	2	1.16	33	1.41	0.3	0.42
3	3	1.159	33	1.73	0.48	0.83
4	4	1.158	33	2	0.6	1.2
5	5	1.158	33	2.24	0.7	1.57
6	6	1.156	33	2.45	0.78	1.91
7	7	1.151	33	2.65	0.85	2.25
8	8	1.149	33	2.83	0.9	2.55
9	9	1.147	33	3	0.95	2.85
10	10	1.148	33	3.16	1	3.16
11	11	1.148	33	3.32	1.04	3.45
12	12	1.147	33	3.46	1.08	3.74
13	13	1.146	33	3.61	1.11	4.01
14	14	1.145	33	3.74	1.15	4.3
15	15	1.145	33	3.87	1.18	4.57
......						

A. *Embedded C Implementation of Linear Regression*

Linear mixed-effects model contains fixed and random effects which are used to calculate multiple variabilities in related measurements. It is defined as $y = \boldsymbol{\beta_0} + \boldsymbol{\beta_1} x + \epsilon$, where y is the dependent variable which is determined by the x which is the independent variable. β_1 describes the slope of the line and β_0 is the intercept. Variable ϵ is a random variable which is out of control in this environment or random fluctuations. The assumptions to the random variable ϵ are: E(ϵ)=0 for all situations; and Var(ϵ)=σ^2. The model is an assumption which is determined by β_0 and β_1. The estimation of β_0 and β_1 is based on the value of y and x.

The multiple regression for ith observation can be described as $y_i = \boldsymbol{\beta_0} + \boldsymbol{\beta_1} x_{i1} + \boldsymbol{\beta_2} x_{i2} + \ldots + \boldsymbol{\beta_i} x_{ij} + \ldots + \boldsymbol{\beta_m} x_{im} + e_i,\ i = 1, \ldots, n,\ j = 1, \ldots, m$. The result of the observation y_i is determined by different independent variables, x_{i1}, x_{i2},.... x_{im}. The assumptions of this model are: E(e_i) = 0; and Var(e_i) = σ^2. The model is also equivalent to:

$$\mathrm{y} = \mathbf{X}\beta + \epsilon \tag{1}$$

where

$$\mathbf{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1m} \\ 1 & x_{21} & x_{22} & \cdots & x_{2m} \\ 1 & x_{31} & x_{32} & \cdots & x_{3m} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{nm} \end{bmatrix} \tag{2}$$

$$\beta^T = \begin{bmatrix} \beta_0 & \beta_1 & \beta_2 & \cdots & \beta_m \end{bmatrix} \tag{3}$$

$$\mathrm{y}^T = \begin{bmatrix} y_1 & y_2 & \cdots & y_n \end{bmatrix} \tag{4}$$

The $\mathbf{X}$ is $n \times (m+1)$ matrix which is independent variables where n is the number of equations and $(m+1)$ is the number of terms in one equation. This means the number of factors which influence the result is $(m+1)$ and the number of observations is n. All of the members inside of the matrix are derived from Eq. (1). The first column is equal to 1 which means the first term of the equation is constant. The regression coefficients β is determined by minimizing the sum of squares of deviations of y. The $\hat{\beta_0}$,$\hat{\beta_1}$,...,$\hat{\beta_m}$ is obtained by minimizing:

$$\sum_{i=1}^{n} \hat{\epsilon}_i^2 = \sum_{i=1}^{n} (y_i - \hat{y_i})^2 \tag{5}$$

$$= \sum_{i=1}^{n} (y_i - \hat{\beta}_0 - \hat{\beta}_1 x_{i1} - \hat{\beta}_2 x_{i2} - ... - \hat{\beta}_m x_{im})^2 \tag{6}$$

The value of $\hat{y_i}$ is the estimation of y_i which is not equal to y_i. The common way to find the $\hat{\beta_0}$,$\hat{\beta_1}$,...,$\hat{\beta_m}$ is to minimize the Eq. (5) which is usually to differentiate the whole equation with respect to β. Then, let the result equal zero. But, we can also write the equation in matrix notation which will express more compactly. The equation can be written as:

$$\hat{\epsilon}'\hat{\epsilon} = (\mathsf{y} - \mathbf{X}\hat{\beta})'(\mathsf{y} - \mathbf{X}\hat{\beta}) \tag{7}$$

From Eq. (7), the following can be obtained:

$$\hat{\epsilon}'\hat{\epsilon} = \mathsf{y}'\mathsf{y} - 2\mathsf{y}'\mathbf{X}\hat{\beta} + \hat{\beta}'\mathbf{X}'\mathbf{X}\hat{\beta} \tag{8}$$

The the value of $\hat{\beta}$ can be calculated from differentiating $\hat{\epsilon}'\hat{\epsilon}$ with respect to $\hat{\beta}$:

$$\frac{\partial \hat{\epsilon}'\hat{\epsilon}}{\partial \hat{\beta}} = 0 - 2\mathsf{y}'\mathbf{X} + 2\hat{\beta}'\mathbf{X}'\mathbf{X} = 0 \tag{9}$$

The normal equation is in Eq. (10):

$$(\mathbf{X}'\mathbf{X})\hat{\beta} = \mathbf{X}'\mathsf{y} \tag{10}$$

If $\mathbf{X}$ is full-rank, the $\mathbf{X}'\mathbf{X}$ in nonsingular, and Eq. (10) can be written as:

$$\hat{\beta} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathsf{y} \tag{11}$$

It is used to obtain the least-square estimator $\hat{\beta}$. $\hat{\beta}$ is called lead-square estimator. $\hat{\beta}$ is the linear function of y. After the value is determined, it can be used to estimate the variable by given values.

In this implementation, we collect data from four batteries. The distribution of β of four batteries follows normal distribution. Once the mean and variance of β of the four batteries are determined, a random β value can be generated from the distribution. Because simple linear regression is used for comparison, the goal is to find the estimated values of $\hat{\beta}_0$ and $\hat{\beta}_1$ to generate β_0 and β_1.

The process can be summarized as minimizing the sum of squared residuals $\hat{\epsilon}_i$:

$$\hat{\epsilon}_i = y_i - a - bx_i \tag{12}$$

In other words, the $\hat{\alpha}$ and $\hat{\beta}$ are to designate to solve:

$$\begin{gathered} \min_{a,b} Q(a,b) \\ Q(a,b) = \sum_{i=1}^{n} \hat{\epsilon}_i^2 = \sum_{i=1}^{n} (y_i - a - bx_i)^2 \end{gathered} \tag{13}$$

We can derive the value of a and b which minimize the objective function Q [11]:

$$\hat{\alpha} = \bar{y} - \hat{\beta}\bar{x} \tag{14}$$

$$\begin{aligned} \hat{\beta} &= \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2} \\ &= \frac{\mathrm{Cov}(x,y)}{\mathrm{Var}(x)} \\ &= r_{xy}\frac{s_y}{s_x} \end{aligned} \tag{15}$$

where $\bar{x}$ and $\bar{y}$ is the average of x and y. Var(x) and Cov(x,y) are the sample covariance and sample variance of x and y, respectively. And r_{xy} is the sample coefficient between x and y.

Then, if the expression is substituted by:

$$f = \hat{\alpha} + \hat{\beta}x \tag{16}$$

We can get:

$$r_{xy} = \frac{\bar{xy} - \bar{x}\bar{y}}{\sqrt{(\bar{x^2} - \bar{x}^2)(\bar{y^2} - \bar{y}^2)}} \tag{17}$$

The determination coefficient (R^2) can come from the Eq. (17) which can be used to measure how the data are close to the regression line.

The method to generate the random number based on the distribution of β is introduced by George Marsaglia [12] which is called the polar method. It is similar to Box-Muller transformation which is a pseudo-random number sampling method for generating standard normal distribution random numbers [13]. This method is proposed by George Edward Pelham Box and Mervin Edgar Muller which is wildly used in computer science and statistics. This method defines two independent random variables which are distributed between 0 and 1. There are two basic forms constructing the whole method:

$$Z_0 = R\cos(\theta) = \sqrt{-2\ln U_1}\cos(2\pi U_2) \tag{18}$$

$$Z_1 = R\sin(\theta) = \sqrt{-2\ln U_1}\cos(2\pi U_2) \tag{19}$$

where Z_0 and Z_1 are both following standard normal distribution.

To avoid using trigonometric function to reduce computation cost on ECS, Z_0 and Z_1 can be rewritten by using polar form proposed by R. Knop in [14]:

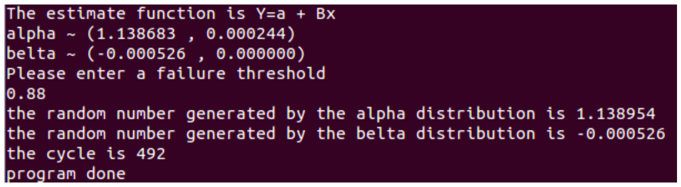
$$Z_0 = R\cos(\theta) = \sqrt{-2\ln U_1}\cos(2\pi U_2) \tag{20}$$

$$= \sqrt{-2\ln s}(u/\sqrt{s}) = u\sqrt{-2(\ln s)/s} \tag{21}$$

$$Z_1 = R\sin(\theta) = \sqrt{-2\ln U_1}\cos(2\pi U_2) \tag{22}$$

$$= \sqrt{-2\ln s}(v/\sqrt{s}) = v\sqrt{-2(\ln s)/s} \tag{23}$$

Then, the random numbers are generated, the model can be used to predict the batteries' cycle to failure. Fig. 6 shows the cycles to failure prediction on a Xilinx ZC-706 SoC-FPGA based ECS. The model can be transferable to any Linux on embedded systems.



```
The estimate function is Y=a + Bx
alpha ~ (1.138683 , 0.000244)
belta ~ (-0.000526 , 0.000000)
Please enter a failure threshold
0.88
the random number generated by the alpha distribution is 1.138954
the random number generated by the belta distribution is -0.000526
the cycle is 492
program done
```

Fig. 6. Cycles to Failure Prediction on SoC-FPGA based ECS

V. Conclusion

The paper introduces our work-in-progress. We have evaluated and assessed two different boot sequences on SoC-FPGA. We will keep investigating other booting sequences to secure ECS on SoC-FPGA. A linear regression based prognostic algorithm has been implemented in embedded C on the Xilinx ZC706 SoC-FPGA based ECS to predict the remaining useful life of batteries. However, ZC706 SoC-FPGA does not support reverse boot. Therefore, we will test the same algorithm on Intel DE1-SoC FPGA which supports reverse boot and compare the performance. Then, more sophisticated prognostic algorithms that we have proposed in [10] will be implemented on the platform to develop a secure, reliable, and portable prognostic device. The device is scalable to predict the health condition of larger scale engineering systems to optimally manage and maintain the engineering systems.

References

[1] D. Moller, *Guide to Computing Fundamentals in Cyber-Physical Systems*. Springer, 2016.

[2] A. Massa and M. Barr, *Programming Embedded Systems*. O'Reilly Media, Inc., 2006.

[3] B. Barney, 2018, available at https://computing.llnl.gov/tutorials/parallel_comp/#Whatis.

[4] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, H.Yang, and W. Dally, "Ese: Efficient speech recognition engine with sparse lstm on fpga," 2017.

[5] E. Nurvitadhi, G. Venkatesh, J. Sim, D. Marr, R. Huang, J. Ong Gee Hock, Y. T. Liew, K. Srivatsan, D. Moss, S. Subhaschandra, and G. Boudoukh, "Can fpgas beat gpus in accelerating next-generation deep neural networks?" in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17. New York, NY, USA: ACM, 2017, pp. 5–14. [Online]. Available: http://doi.acm.org/10.1145/3020078.3021740

[6] P. Schaumont, I. Verbauwhede, K. Keutzer, and M. Sarrafzadeh, "A quick safari through the reconfiguration jungle," in *Proceedings of the 38th Annual Design Automation Conference*, ser. DAC '01. New York, NY, USA: ACM, 2001, pp. 172–177. [Online]. Available: http://doi.acm.org/10.1145/378239.378404

[7] K. Compton and S. Hauck, "Reconfigurable computing: A survey of systems and software," *ACM Comput. Surv.*, vol. 34, no. 2, pp. 171–210, Jun. 2002. [Online]. Available: http://doi.acm.org/10.1145/508352.508353

[8] Online, *RocketBoards.org*.

[9] *Using Linux on the DE1-SoC*. Altera Corporation, 2016, available at ftp://ftp.altera.com/up/pub/Altera_Material/16.0/Tutorials/Linux.pdf.

[10] J. Guo, Z. Li, and M. Pecht, "A bayesian approach for li-ion battery capacity fade modeling and cycles to failure prognostics," *Journal of Power Source*, vol. 281, pp. 173–184, May 2015.

[11] J. Kenney and E. S. Keeping, "Linear regression and correlation," *Mathematics of Statistics, 3rd ed. Princeton, NJ*, pp. 252–285, 1962.

[12] G. Marsaglia and T. Bray, "A convenient method for generating normal variables," *SIAM Review*, vol. 6, no. 3, pp. 260–264, 1964.

[13] G. E. P. Box and M. E. Muller, "A note on the generation of random normal deviates," *Ann. Math. Statist.*, vol. 29, no. 2, pp. 610–611, 1958.

[14] R. Knop, "Remark on algorithm 334 [g5]: Normal random deviates," *Communications of the ACM*, vol. 12, no. 5, p. 281, 1969.