Energy-aware Timing Analysis of Intermittent Execution

Experimentation and Evaluation Plan

Ferhat Erata Sinan Yıldırım Arda Göknil Ruzica Piskac Jakub Szefer

28 February 2021

1 Testbed Applications

To demonstrate the usefulness of the approach, we aim at developing one or two sensing applications. The first candidate is anomaly detection through audio sensing.

The other candidate that we discussed is motion detection similar to anamoly detection. In these two scenarios, we don't plan to connect a real sensor or a transmitter to the launchpad. Instead, we will simulate the energy consumption and timing by delays and timers as well as the data received or transmitted through embedding sample or generating random data.

Apart from sensing application, we can elaborate the evaluation on security applications using computational intensive tasks such as data encryption, decryption and hashing algorithms if we have enough time or we can analyze security in the context of intermittent execution in a follow-up work.

We composed a set of `Tasks` that are frequently used in resource constrained embedded system development. They are open-source implementation of the following algorithms: *FFT*, *JPEG*, *SHA*, *RSA*, *AES*, *CRC*, *QSort*, and *bitcount*. We managed to successfully deploy and run those algorithms on the MSP430FR5994 launchpad.

Task	Explanation	Checkpointed	Measured	Analyzed	Exp. Timing
FFT	Fast Fourier Transform	✓			3/s
JPEG	Jpeg image compression				
SHA	Secure Hash Algorithm	✓			10/s
CRC	Cyclic Redundancy Check	✓			
AES	Advanced Encryption Standard	✓			
RSA	RSA public-key cryptosystem				
QSort	Quicksort Algorithm				
BitCount	Bit Count Algorithms				

The `compute()` part of the embedded application is the stage of the application where we assume the embedded programmer places checkpoints. The `ETAP` will be able to infer the probability distribution of the timing of `compute()` stage, from sensing to transmit; it may also report the rate of successful runs of the application (such as the 68%, 95% and 99.7% success rates); or it may check whether a timing constraint is satisfied or not within a confidence interval.

Testbed Application	Tasks	Checkpointed	Measured	Analyzed
Audio Sensing & Anomaly Detection [1.1]	FFT,			
Encrypted Communication [1.2]	SHA, AES			

1.1 Sensing Applications

```
Sense() -> Compute() -> Transmit()
```

- *FFT*: Fast-Fourier Transform.
- *JPEG*: Graphic Image Compression.

Application 1: Audio Sensing and Anomaly Detection

1. Sample Audio (Embedded test data or random-generated data).
2. Perform *FFT* (Time to Frequency)
3. Detect an event (anomaly): The frequency of the sample exceeds the threshold.

4. Send (raise a pin) it in 10 seconds or `Green (normal) Red (anomaly)`
5. Go to 1st step.

Application 2: Motion Detection

1. Capture 1st Frame (RGB data format) 640x480 RGB array -> JPEG array
2. Hash frame 1 *SHA* (or *jpeg* encoder -compression-)
3. Capture 2nd Frame (RGB data format)
4. Hash frame 2 *SHA* (or *jpeg* encoder -compression-)
5. Compare hashes (or compare jpegs)
6. Transmit (if the anomaly detected, blink the LED twice)
7. Go to 1st step.

1.2 Security Applications

`Receive() -> Compute() -> Transmit()`

- *RSA* asymmetric encryption (public/private key)
- *AES* symmetric encryption (shared key)
- *SHA* authentication, hashing
- *CRC* Cyclic Redundancy Check (CRC)

Application 1: Authentication (Access control)

1. Wait for receiving a signal (when pushed a button send a random secret message from an array)
2. *SHA* (check the signature, if it is authentic, blink the LED)
3. Signal

Application 2: Encrypted Communication

1. Wait for receiving a signal
2. Analyze()
3. *AES* (Encrypt)
4. Transmit
5. Go to 1st step.

Application 3: Error Correcting of signed messages

1. Wait for receiving a message
2. *CRC* (checksum)
3. analyze the message
4. sign the signal
5. send the signal (blink LED)
6. Go to 1st step.

2 Hardware Setup

2.1 Checkpointing inside Tasks

Batteryless devices, instead of getting energy from the power grid or a battery, harvest their energy from their environment (e.g., sunlight or radio waves). Energy harvesting devices replace batteries with a small energy storage capacitor. We can consider the size of the capacitor as being proportional to the duration of computation.

We run the *SHA* task successfully using the checkpointing library that Sinan's PhD Student Eren Yildiz developed. We specified a `fixed number of cycles` and set a timer duration to fire soft reset to simulate `power failures`. After soft reset, we also set a `random delays` simulating the recharge period of the capacitor. After a power failure, with the minimum overhead, the runtime restores the execution from the last checkpointed region. During the computation we turn on the *LED1* (Red) and during the power failure we turn the LED1 off.

In the application development, one of the challenges is to place checkpoints (such as in *FFT* and *JPEG* tasks) carefully so that we should maintain forward progress. The total number of cycles executed between two consecutive checkpointed region should always consume less energy than the maximum stored energy.

So I might cite some things [Shea and Boldt, 2014]

2.2 Considerations for Clock Speed

1 MHz or 8 MHz

Assuming your CPU frequency is 1Mz , each cycle takes 1uS. so for 1 second you have to use the count 1000000.

There is a direct tradeoff between clock speed and energy consumption. What should we set for the clock speed? 8MHz?

2.3 Device Setup

- In our evaluation, we use TI's msp430FR6989 development board, the latest released version of TI boards that includes an embedded byte addressable persistent memory. The microcontroller is a 16-Bit RISC architecture and is clocked at 8-MHz. It has 2KB of SRAM or volatile memory, and an effective 82KB of Ferroelectric RAM (FRAM) – a non-volatile, byte-addressable memory. The 82KB of FRAM is shared between the application code and application data allocated as persistent. The memory model is flat, meaning a single contiguous address space is used and the CPU directly addresses SRAM, FRAM, etc. The CPU supports a 20-bit addressing mode (msp430x) required for accessing the full address space, which we activate by default for our experiments.
- We applied CleanCut to real code on real energy-harvesting hardware. We used the WISP energy-harvesting device, which has an 8MHz MSP430FR5969 MCU with 64KB of non-volatile memory and a 47μF capacitor. We powered the WISP wirelessly using a ThingMagic Astra-EX RFID reader at 16 dBm. We fixed the WISP 45 cm from the power antenna, parallel to its surface.
- The WISPCam's MSP430FR5969 microcontroller has a 64KB on-chip FRAM memory that supports 48 Mb/s data transfer from the camera to the microcontroller while burning only a few milliwatts of power. WISPCam main board, based heavily on the WISP 5.0. An AVX BestCap series supercapacitor allows the WISPCam to accumulate the 20 mJ necessary for image capture and transfer. we found that at least 10 mJ of energy is required to capture and store a 176*144 pixel (QCIF) gray scale image such as those seen in Figure 12 (Images with higher contrast or higher resolution will require expending more energy). We select a 6.08 mF supercapacitor and target 20 mJs of usable energy.
- We implement Ratchet using the LLVM compiler infrastructure. Beyond verifying that Ratchet output executes correctly on an ARM development board, we build an ARMv6-M energy-harvesting simulator,

with wholly non-volatile memory. We use an average lifetime of 100 ms to match the setup of previous works. With an average lifetime of 100 ms running with a 24 MHz clock, this gives us a mean lifetime of 2,400,000 cycles. Before each bout of execution, the simulator samples from a Gaussian whose mean is the desired mean lifetime (in cycles) and uses that value as the number of clock cycles to execute for before inducing the next power cycle. To simulate a power cycle, we clear the register values.

2.4 Energy Harvesting Simulator on MSP430

An energy-harvesting simulator is required because of the difficulties associated with developing and debugging intermittently powered devices. Using a cycle accurate simulator we are able to simulate failures with a probability distribution that can model the true frequency and effects of power failures experienced by devices in the real world. Using a simulator also allows us to take precise measurements of how much progress a program makes per power cycle, and the cycles consumed by re-execution.

3 Statistical Inference of the Instruction-based Energy Models

We should infer the probability density function of the energy consumption of each instruction types. Instructions work in MSP430 ISA with different operating modes. Therefore, the energy trace collection should also consider addressing modes of the instructions.

3.1 Energy Traces collected from the EnergyTrace++ of Code Composer Studio.

We aim to profile about 20 different instruction, each of them should run at least 100 times. The problem is there is no commandline interface of API for us to write a script. Each test requires us to click a set of buttons on the Eclipse IDE (CCS Studio). We are working on a python script that automated mouse click events.

Here is an example trace data. We can also measure specific execution intervals through the graphical user interface of the Code Composer Studio.

What is the causal relationship here? What is the explanatory (x:independent), response(y:dependent variable)

```
Time (ns),Current (nA),Voltage (mV),Energy (uJ),DSR
"0"      ,"3356900"   ,"3287"      ,"50.1"     ,"0x3E1017E80000830"
"902000" ,"3356900"   ,"3272"      ,"60.5"     ,""
"1744000","3356900"   ,"3277"      ,"70.1"     ,""
"2588000","3356900"   ,"3272"      ,"79.2"     ,""
"3431000","3356900"   ,"3276"      ,"88.8"     ,""
"4530000","3410900"   ,"3273"      ,"101.1"    ,""
...
```

3.2 Sampling ADD instruction

Sample the instructions wrt. addressing modes.

1. Measure the energy consumption E_{total}

```
// start the energy profiling here
while(1000){
    ...
  __asm__ __volatile__ (" ADD R3,0(R15)");
}
// stop the energy profiling on halt
```

2. Measure the energy consumption $E_{overhead}$

```
// start the energy profiling here
while(1000){
 ; // noop instruction
}
// stop the energy profiling on halt
```

$E_{ADD} = (E_{total} - E_{while} - E_{startup})/1000$

3. Repeat sampling x100.

uj (one decimal precision)

1. create a vector from samples

```
add <- c(11.3, 12.5, ...) # list of 100 samples
```

2. fit a normal distribution

```
## reproducible example
set.seed(0); x <- rnorm(100)
## using MASS

library(MASS)
fit <- fitdistr(x, "normal")
class(fit)
```

```
## [1] "fitdistr"
```

```
# [1] "fitdistr"

para <- fit$estimate
#         mean            sd
#-0.0002000485  0.9886248515

hist(x, prob = TRUE, breaks = 20, ylab = "", xlab = "", main = "")
curve(dnorm(x, para[1], para[2]), col = 2, add = TRUE)
```

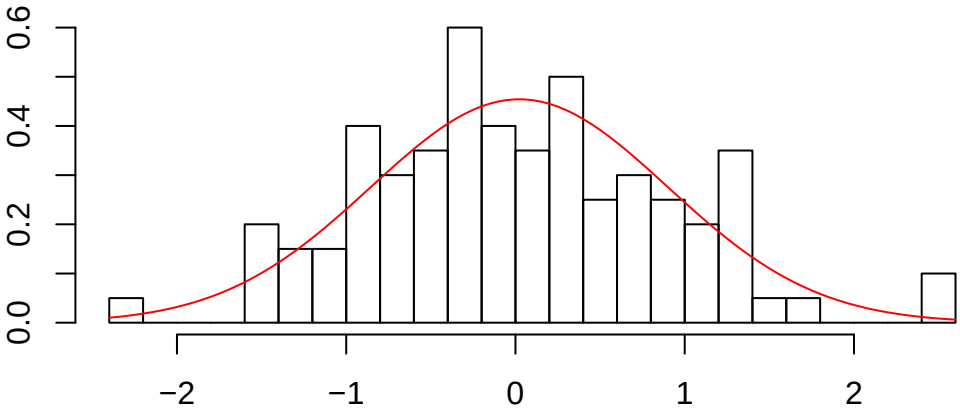

Figure 1: The instruction-based energy model for Add instruction.

3.3 Sampling MOV instruction

there are several modes to consider:

- store to volatile ram (SRAM)
- store to nv ram (FRAM)
- store to register

maybe this will lead to a bimodal distribution

4 Other methods to derive instruction-based energy models

4.1 Optimization Model

Although the previous approach may successfully infer probabilistic models for each instruction types, they are still MPS430 ISA. However, the probabilistic symbolic execution engine only runs on LLVM IR. There is no one-to-one correspondence with the architecture independent representation of the instruction set, and the MSP420 instruction set. Nevertheless, we can construct an optimization problem as the following: first we can collect samples from a set of benchmarks applications; for each program, we can sample 100 and take its expectation; and we can count the frequency of each instruction type; and finally construct linear equations

Here is an R program solving linear equations

```
#https://cran.r-project.org/web/packages/matlib/vignettes/linear-equations.html
rm(list = ls(all.names = TRUE)) #will clear all objects includes hidden objects.
library(matlib)   # use the package
A <- matrix(c(1, 1, 1,
              1, 2, 1,
              2, 1, 1), 3, 3, byrow = TRUE)
colnames(A) <- c("alloca", "load", "store") # "io.read", "io.write"
rownames(A) <- c("program.1", "program.2", "program.3")
A
```

```
##           alloca load store
## program.1      1    1     1
```

```
## program.2      1    2     1
## program.3      2    1     1
```

```
b <- c(21, 31, 31) #1
showEqn(A, b)
```

```
## 1*x1 + 1*x2 + 1*x3  =  21
## 1*x1 + 2*x2 + 1*x3  =  31
## 2*x1 + 1*x2 + 1*x3  =  31
```

```
Solve(A, b, fractions = TRUE)
```

```
## x1      =  10
##   x2    =  10
##     x3  =   1
```

```
b <- c(21, 31, 32) #2
showEqn(A, b)
```

```
## 1*x1 + 1*x2 + 1*x3  =  21
## 1*x1 + 2*x2 + 1*x3  =  31
## 2*x1 + 1*x2 + 1*x3  =  32
```

```
Solve(A, b, fractions = TRUE)
```

```
## x1      =  11
##   x2    =  10
##     x3  =   0
```

```
b <- c(21, 31, 33) #3
showEqn(A, b)
```

```
## 1*x1 + 1*x2 + 1*x3  =  21
## 1*x1 + 2*x2 + 1*x3  =  31
## 2*x1 + 1*x2 + 1*x3  =  33
```

```
Solve(A, b, fractions = TRUE)
```

```
## x1      =  12
##   x2    =  10
##     x3  =  -1
```

Here the same procedure is modeled as a linear programming problem. The configuration is different.

```
# Linera Programming
# https://rstudio-pubs-static.s3.amazonaws.com/534936_8eeb46b4b20d47509e4dede705dbb1c4.html
library(lpSolveAPI)

lps.model <- make.lp(0, 3) # define 3 variables, the constraints are added below
```

```
add.constraint(lps.model, c(6, 2, 4), "<=", 150)
add.constraint(lps.model, c(1, 1, 6), ">=", 0)
add.constraint(lps.model, c(4, 5, 4), "=", 40)
# set objective function (default: find minimum)
set.objfn(lps.model, c(-3, -4, -3))
# write model to a file
write.lp(lps.model, 'model.lp', type = 'lp')

# these commands defines the model
# /* Objective function */
#   min: -3 C1 -4 C2 -3 C3;
#
# /* Constraints */
# +6 C1 +2 C2 +4 C3 <= 150;
# +  C1 +  C2 +6 C3 >=   0;
# +4 C1 +5 C2 +4 C3  =  40;
#
# writing it in the text file named 'model.lp'
solve(lps.model)

# Retrieve the var values from a solved linear program model
get.variables(lps.model)  # check with the solution above!
```

4.2 Instruction-based Model from Block-based Mapping.

Cemal hoca's method

4.3 Does this gap pose another research question?

Recent advances in security testing of embedded and IoT software focus on research having built on top of tools that rely on LLVM compiler infrastructure. While this technology is powerful and allows researchers to experiment, relieving them from implementation issues. Yet, these solutions might be difficult to apply in embedded and IoT domains which adopt different (and mostly proprietary) compiler and OS technologies. Investigate binary analysis ?? what are the drawbacks of binary analysis?

5 Statistical Inference for the Incoming (Harvested) Energy Model

We need to infer a probability density function for the harvested energy. It will be our stochastic incoming energy model as an input of the probabilistic symbolic execution engine.

I haven't investigated yet how to measure the voltage pin???

- We need to prepare a meaningful setup: 1 - 3 - 5 meters collect profiles; maybe 45 degrees. However, if the application's context is gunshot detection, then we need to at least 50 meters???

6 Resource Planning

1. sampling and statistical inference:

- instruction-based energy model (`1.5 week`) : `Enes, Gokçin`
- incoming (harvested) energy model (`1.5 week`) : `Ferhat`

2. application development

- audio anomaly detection (`1 week`): `Enes, Gökçin`
- motion detection (`1 week`): `Enes, Gökçin`

3. intermittent execution semantics

- development of the missing parts of the intermittent execution semantics (`1 week`): `Ferhat`
- evaluating the probabilistic symbolic execution with the algorithms used in the applications; if necessary, introduce ways to increase the scalability such as state merging (`1 week`): `Ferhat`

4. Evaluation

- final evaluation with the models (`1 week`) : `Enes, Gökçin`

5. Specification of the timing constraints

- development (`1 week`) : `Ferhat`

Minimum Effort

Ferhat: 4.5 weeks

Enes, Gökçin: 4.5 weeks

References

Nicholas Shea and Annika Boldt. Supra-personal cognitive control. *Trends in Cognitive Sciences*, 18:186–193, 2014. doi: 10.1016/j.tics.2014.01.006.