A Chain-based Approach to Software Optimization in Low-Power Energy-Harvesting Intermittent Computing Devices for Agricultural Health Monitoring

Yaxin Deng
Viterbi Engineering Department
University of Southern California
yaxinden@usc.edu

Eric Escudero
Viterbi Engineering Department
University of Southern California
eescuder@usc.edu

Miguel Gallegos
Viterbi Engineering Department
University of Southern California
gall563@usc.edu

Zikun Yang
Viterbi Engineering Department
University of Southern California
zikuny@usc.edu

Shanin Nazarian Ph.D
Viterbi Engineering Department University of Southern California
shahin.nazarian@usc.edu

A. Cover page

PROBLEM DESCRIPTION

Intermittent devices are those that run solely on the energy harvested from the environment. Due to the randomness associated with the environment, programs may not run as expected in these devices. It is therefore necessary to have software tools to correct or maintain data consistency, concurrency, control-flow, and communication between devices. Our software tools provide a framework for simulating environmental conditions related to agriculture. The simulation environment is built using a library written in C++ that we implemented from scratch based on the Chain methodology created by Brandon Lucia and his students. For the following phase (3), we will implement a debugging tool and, if time permits, a GUI for users to make the most from the simulation environment. We also hope to provide a plotting tool for analyzing trends in intermittent behavior.

PROJECT TIMELINE

This project was scheduled to a three phase timeline. In Phase I, our team provided analysis of problems in terms of memory, runtime, power, and started designing the simulation platform that is capable of simulating chain-base algorithms to be used to address the issues listed above. Meanwhile, we made decisions based on the project application and details regarding implementation. To better show the implementation of energy harvesting devices, our project decided to pick the Agricultural Health Monitoring application for intermittent computing since it provides opportunity to contribute to an environmentally important field that benefits a wide range of communities all over the world. Based on this application, we also decided the software implementation for better simulate the realistic situation which require separate I/O devices to collect data from environment, The Python package, wxPython, was chosen in order to help alleviate the workload of creating a GUI that will allow the user to plot the data and analyze the results.

Phase 2 was testing the original *chainlib* package from the paper, and testing the implementation on c++ library. Our team analyzed the Chain library and debugger tool (EDB) which was downloaded from GitHub, it was determined that the implementation of original *chainlib* need both software(including TI GCC MSP430, LLVC/Clang toolchains) and MSP430 hardware support. Therefore, the Chain library was restructured and implemented in our own way in c++ to fit both the application and timeline we desired. Meanwhile, We achieved this by implementing the Chain-based library and methods and creating a simple test to determine if our code works as intended. This c++ chain-based library was 90% built in phase II, and implementations of the energy-harvesting application contains Tasks-control and Channels-control was implemented completely in software.

Completion of the project will be done in Phase 3. Most of the function is recreated in a C++ library. Some minor revisions to the software implementation may occur, but in large part, finalizing the debugging and testing portion of the code will be the main focus. If time permits, a GUI will also be included to give a more formal presentation of our software application. Section B will be updated with the current experimental framework and results, and an evaluation algorithm will be used to verify these results.

B. Report

a. Abstract

Abstract*— Intermittent devices have a unique place in the low-power device space because they are batteryless. This comes with benefits and consequences. The purpose of this paper is to introduce intermittent systems along with the inherent complications, to propose a framework for simulation environments for intermittent devices to test program structure and model intermittent behavior based on varying inputs, to propose algorithmic changes to Chain-based methodology to improve performance, and to evaluate the framework using probabilistic modeling. The paper will finish with a conclusion summarizing the findings and takeaways.

b. Introduction

Intermittent computing devices are batteryless and harvest energy from the environment to execute programs [1]. The harvested energy can come from a variety of sources including sunlight, vibrations or another thermal source, and RF waves [1]. Due to the intermittent nature of such devices, the execution of programs running on these devices becomes unreliable and sometimes unusable. This can happen for several reasons including non-termination loops where one part of the program continues to execute after power failures for being too energy expensive and the device never having the required energy to run it completely, having incorrect values for variables due to values being overwritten before a power failure, and having data become stale after a long shut down [1]. Chain, a C-library and runtime, was created to help deal with these problems. Chain works by having code blocks of a program called tasks that are executable to completion given a certain threshold of energy harvested. The tasks can store data in channels that are made up of non-volatile memory which allows another task on the other end of the channel to use the same data when they are executing. To work on this project, the team researched Chain and its methods. In addition to this, the team also researched common functions for agricultural-based embedded systems. This was to help plan the code to be written for the simulation environment. The simulation environment is based on Chain, but is written for C++ and is not intended for use with hardware, but instead only in software. The novelty of our framework is that it can allow users to simulate intermittent behavior given specific or random parameters. In addition to this, our framework has a user interface for easy use along with a debugging tool for determining different types of errors in their programs. Additional functions can be added by developers to test unique functionality and performance for non-trivial programs for intermittent devices. Lastly, timing analysis tools are also available with this framework for tracing to notable events or for comparing performance to existing software tools for intermittent devices.

c. Framework

Framework Description

In order to faithfully emulate the *libchain* library, the functionality must remain the same. To do this, both tasks and channels had to be implemented. The task functionality can be described as functions that execute while the device is on and that are connected to each other. The channel functionality can be described as non-volatile data that is exclusive to two Tasks. Thus, for every N Tasks, there should be N ways to access non-volatile data through channels. Based on the research from the Chain paper and *libchain* library, we implemented a C++ program as a simulator to recreate the program interface between tasks and memory allocation for channels. As shown in Figure 1, our C++ program creates a task-based control-flow, and exchanges data only between channels. For achieving the functionality of a Chain-based intermittent program without hardware support, we implemented a linked list to represent the FRAM memory as shown in the dark green portion of Figure 1. We then created a linked-list type memory table in a global scope to allow data to be written or read from a table location dependent on the channel that the current memory allocation calls for. Each task's data can store the most recent 10 sets of the data created on the channel for data recording purposes. For the task control flow, we split the function on the traditional program to multiple tasks, and each task will go to the next task chosen by conditions within the recently finished task. Eventually there will come a task that sends output signals for turning on or turning off an external device. In the case of our agricultural-based tasks, for turning on or off fans, irrigation systems, and sprinklers. A channel is created due to the interaction between tasks like the blue or red section inside Figure 1 and each channel can only be allocated the memory at the proper position (e.g. Channel 1,2 circle in red can only access data in task 1 and 2 memory section). The read and set functions for this design will still be O(n) complexity due to the linked-list structure. Origin task is a non-volatile memory variable and it represents the task that needs to be compiled after a power restore. Once a task successfully completes, the origin task is updated and points to the next task that needs to be compiled.

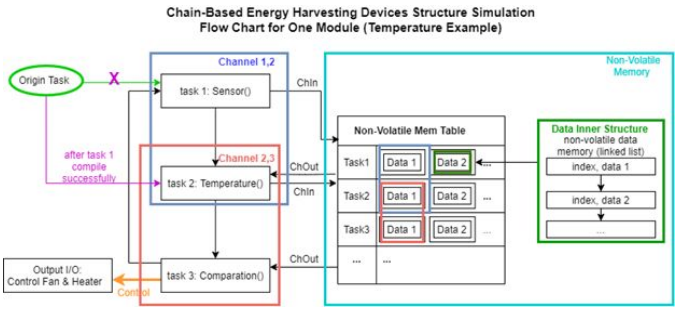

Figure 1: Chain-Based Energy Harvesting Devices Intermittent Program Structure Simulation Created by C++ Language. Diagram using temperature as an example for our agriculture application.

Illustrated in Figure 2 is an example for measuring the temperature using a sensor. We create three tasks: task Sensor() to measure number from temperature sensor three times; task Average() to calculate the average number from the measurement; and task Output_I/O() to compare with the temperature boundary and send a signal to I/O devices. Each time we enter a task, we read the current index number that the memory is stacking on, then run its function depending on the requirement. The set function will describe the position that the memory channel is going to be written to and limit the member's access control. For the read function, the non-volatile memory storage inside the channel will be accessible and stored in volatile memory inside the task before it finishes the task. After a function compiles successfully, the set_origin() function will pass the origin to the next task, and if the power fails, the origin task will be the one that needs to be compiled first. In our design, we store all the tasks in one control flow as a class Task, and the task will be initialized only once after power recovery.

To simulate the power failures that occur in actual intermittent devices, a timer program is used. The timer comes from a GitHub repository and it is used for running the emulation for a specified amount of time. To ensure that the timer works alongside the executing tasks, the tasks are called from within the timer. The timer itself will run on a separate thread from that of the main function. For accuracy in energy-harvesting behavior, there is also a timer for when the device experiences a power failure. The duration of these timers is determined by a sawtooth waveform generated in a Python script. This sawtooth waveform is generated using pseudo-random values, which is representative of the unpredictability associated with the different environment.

Figure 2: An Example pseudo-code to describe the Temperature function control loop.

Random Waveform Model

The numbers used to generate the sawtooth waveform are pseudo-random. It is therefore up to the user to determine the boundaries for which an environment may represent.
For example, in an environment where there are a lot of trees or shade, it may be more difficult to collect energy from sunlight and therefore, it can be expected for the device to be turned off for a longer period of time. The timer function uses three numbers associated from the waveform to determine when the device is on and off. The first one is the total number of time that the waveform lasts. This, in reality, would be a very large number because an intermittent device is expected to work for a long time undisturbed. The second number is how long the device charges for, and the third number is how many times the recharges and runs tasks. From these three numbers, the duration a device is on and off can be determined.

Analysis of Time and Space Complexities

In the original *libchain*, the program used C language to create a library to support intermittent program execution on TI MSP430FR5949 MCU. Inside the library, each memory store in a non-volatile memory sill is separated and stored in its internal area inside the ELF after it compiles. This library recording the binary number represents the origin of the task and storing the binary number into non-volatile memory at the end of each task. For these tasks, the prologue function utilizes a while-loop to repeat the loop, so eventually, the task define will cause an O(n) time complexity to define. For non-volatile memory data, the program can keep track of the stack position, and give the proper access control data inside one channel. For the data exchanging function, called *ChIn* and *ChOut*, this program uses a search similar to the linked list to find the proper memory location and store the data inside non-volatile memory. This library exchange requires hardware support, and eventually, the hardware data will be stored inside the FRAM by using stack, and time complexity will be O(n) for reading this data. To successfully compile *libchain*, two external toolchains are required: a TI GCC MSP430 package for Arch Linux, and LLVC/Clang toolchain to compile C to assembly.

For our Chain program we use linked lists to represent our non-volatile data in channels. Because of this, our access and search time for non-volatile data during a task is O(n). In addition to this, our insertion and deletion time is O(1). Having these efficient time complexities is useful for our framework as we are emulating hardware. Our task methods such as Sense() and Average() run at O(n) worst case time as they may access non-volatile data from the linked list. Since there is an array that stores all of the linked lists, the space complexity for that is O(n*m), where O(n) is due to the linked lists and O(m) is due to how many rows there are in the array. That being said, our task functions are called one by one for each type of sensor. Since there are three sensors and three tasks for each, the time complexity for one complete iteration of all tasks is O(9*n) which still results in O(n). Also, due to

the way we structured our code and for fast simulation purposes, the tasks can be executed from start to finish on multiple threads. This would help with simulation purposes as there would be more data to work with. This is something worth considering for the future.

d. Implementation and Experimental Result

This project requires the use and development of a variety of models, tools and packages that are available from previous research in the field of Intermittent Computing. This project referenced the Chain library package from the CMU Abstract Research Team (*libchain*) to implement an application for Agricultural Health Monitoring using an energy-harvesting device. In order to implement an agricultural-based application, we utilized three main models including, Temperature, Water and Humidity. Each one of these models contained three tasks, Sensor_RAW, Sensor_AVG and Sensor_IO, for a total of nine tasks overall. A block diagram of the software interface is illustrated in Figure 3.

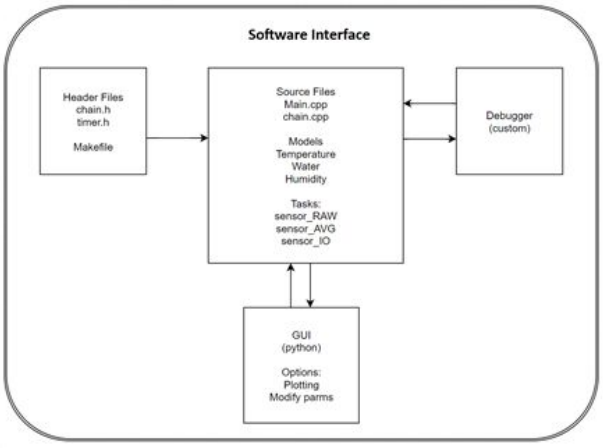

Figure 3: Software Interface Block Diagram

File Architecture and Experimental Setup

Our project requires implementation written completely software. In order to accomplish this, we utilized C++ as our main application code, Python for our testing setup and tool implementation, and a Makefile to assist with compiling all the source code and libraries.

Instead of implementing Chain as a C language extension and a runtime library package, we decided to implement our version of a Chain-based library by rewriting the functions and methods found in *chain.c* (now called *chain.cpp*) and *chain.h* files. This change gave us flexibility in implementation, but still allowed us to utilize the methodology of Chain. The following libraries were utilized within the *chain.h* and *timer.h* header files: chain.h - *iostream*, *vector*, *time*, *random*, *cstdlib*, *ctime*, *fstream*, *unistd*, *tuple*, *stdarg*; timer.h - *iostream*, *thread*, *chrono*.

The Python library packages utilized to implement the front-end and back-end testing of the software application include: *os*, *sys*, *csv*, *random*, *unittest*, *subprocess*, *numpy*, *scipy*, *matplotlib.pyplot*. These library packages allowed implementation of the testing framework for which the Python script was responsible. For example, to generate the random voltage waveform, the *random* package was utilized to select a random range of timing parameters for the waveform. These libraries enabled us to utilize prebuilt classes, functions, etc. in order to implement our version of the Chain-based library.

Software Implementation

The source code we used in our project were implemented in *Main.cpp* and *chain.cpp*. *Main.cpp* contains our *main()* and is the driving code to illustrate our implementation of the energy-harvesting intermittent application. The source file, *chain.cpp*, contains our version of the method implementations of the functions and classes within the header file (*chain.h*).

Our *chain.cpp* source code contains one struct, *Data*, two classes, *Data_nonvol* and *Task*, and multiple class member functions and class members that we utilize to implement the main characteristics of the Chain language, which are *Tasks* and *Channels*. The *Tasks* functionality was implemented within the *Task* class, which contains the member functions *sensor_RAW()*, *sensor_AVG()*, and *sensor_IO()*. These three functions measure the raw sensor data, calculate the average of the raw data, and transmit the saved data to the user interface, respectively. Each of these three tasks has three subtasks that are based on the type of *model* (i.e. temperature, water, or humidity). This was done to decipher between the types of data and give more control over the individual channels. The *Task* class contains class members *model_index*, *task_index*, and *data_str*. These members are utilized to keep track of the model type, task type and data string to be written to the output file, respectively. The remaining member functions within the *Task* class are responsible for such functions as: retrieving the origin task index/model, setting the origin, initializing the *Task* object with a provided model/task index and writing data to an output file.

Since this project was strictly done in software, i.e., no external non-volatile memory was used, the channels were created using the *Data_nonvol* class with the *Data* struct in order to mimic the external non-volatile memory device. The *Data_nonvol* class is utilized to create a linked-list data structure that is able to read and write data elements, of struct type *Data*, to the linked-list. Channels were created between tasks using the *Task* class members, *model_index* and *task_index*. These two members determine which tasks have access to the corresponding data within the non-volatile memory matrix, *Data_Index_Table[][]*, creating a *channel*-like structure between two or more tasks. If the model index and task index do not match for a particular task

that wants to write to the data table, then that task will not have access to this row in the data matrix (i.e. non-volatile memory address space).

In order to implement the power on and power off functionality of the application, three functions were utilized. These functions include *power_on*, *power_off*, and *Timer*. The *power_on* function is responsible for running the main code section (i.e. the code that represents a powered device during nominal operation) and determining when the device should power off, which is based on the provided waveform data that represents the capacitor voltage level timing constraints. Once a power failure occurs, i.e. the V_{hi} voltage threshold is exceeded, the code enters the *power_off* function which is responsible for mimicking an unpowered device by entering a wait state where the main code section (i.e. nominal operation code) is not executing. During this period, data is being dumped from the non-volatile memory matrix for testing purposes. This data is utilized in debugging and verifying accurate implementation of the application code.

Experimental Testing and Results

In order to verify our application, we utilized a Python script to generate a randomized saw-tooth waveform that represents the characteristics of a voltage waveform for the charging device (i.e. capacitor). For simplicity, a saw-tooth waveform was utilized in order to represent the timing characteristics that our code requires. These parameters include the time at which charging starts, the duration of charge (i.e. device off duration), the time at which charging ends (i.e. device turns on and code starts executing), and the duration of the on period. These times represent the voltage thresholds (V_{hi} and V_{lo}) of a charging device that will determine if the device is on or off. Since we do not have actual hardware, these timing constraints, and therefore, voltage threshold levels, can be arbitrarily chosen to mimic a realistic waveform. The particular function used to generate the waveform defaults these to (-1,1). The randomized parameters generate an off duration (charge time) between 70-95 seconds, which results in an on duration (execute time) between 5-30 seconds. The number of total cycles and total time are also randomized between 3-10 cycles (or 300-1000 seconds total). Figure 4 below illustrates a randomized waveform example with 3 total cycles and a charge time of 92s, and discharge time of 8s.

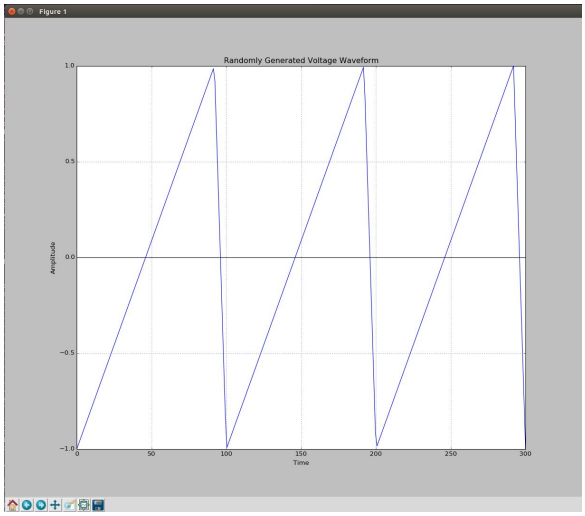

Figure 4: Example of Randomly Generated Saw-tooth Waveform

The Python script then compiles the C++ application and calls the executable with the waveform_data.txt as an argument. After the application executes and performs the tasks within the time constraints determined from the generated waveform, a results.txt file is generated with the output data. This data includes: the task type being executed during a given on period, the data generated within said task (if applicable), and the average data read from the non-volatile memory and whether this data was valid (i.e. within acceptable range regarding the particular model). The Python script then parses through the results.txt file and provides analysis for review by the operator. Below is an example of the results.txt printout that is generated after execution:

```
Executing Sensor RAW Task for Temperature Model
Setting Origin Task to sensor_AVG() for Temperature Model
New origin model is: 0
New origin task is: 1
New Origin Task Set!
---------------------------------------
Executing Sensor AVG Task for Temperature Model
Setting Origin Task to sensor_IO() for Temperature Model
New origin model is: 0
New origin task is: 2
New Origin Task Set!
Raw data is: 57.990612 | 80.280853 | 60.845036
---------------------------------------
Executing Sensor IO Task for Temperature Model
Average data read from non-volatile memory is: 66.372169
**data too low**
Setting Origin Task to sensor_RAW() for Water Model
New origin model is: 1
New origin task is: 0
New Origin Task Set!
---------------------------------------
Executing Sensor RAW Task for Water Model
Setting Origin Task to sensor_AVG() for Water Model
New origin model is: 1
New origin task is: 1
New Origin Task Set!
---------------------------------------
Executing Sensor AVG Task for Water Model
Setting Origin Task to sensor_IO() for Water Model
New origin model is: 1
New origin task is: 2
New Origin Task Set!
Raw data is: 12.093601 | 5.301159 | 48.146919
---------------------------------------
```

Figure 5: Example of Output Results

C. Code Release

Our implementation of Chain-base simulator for agriculture application is included in chain.cpp, chain.h, timer.h. The source code is available at: https://github.com/Mgalleg/Chain-IC

References

[1] A. Colin and B. Lucia. Chain: tasks and channels for reliable intermittent programs. *In Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 514–530. ACM, 2016.

[2]

[3]