AY 2019

Sloth:
A Reconfigurable Compiler for Task-based Intermittent Programming

A Thesis Submitted to
the Department of Computer Science and Communications Engineering,
the Graduate School of Fundamental Science and Engineering of
Waseda University
in Partial Fulfillment of the Requirements for
the Degree of Master of Engineering

By: Zebo Yang (5117FG28 -3)

July 22th, 2019

ADVISOR: PROF. TATSUO NAKAJIMA

Table of Contents

ABSTRACT **1**

CHAPTER 1. INTRODUCTION **2**

CHAPTER 2. RELATED WORK **6**

SECTION 2.1 ULTRA LOW-POWER HARDWARE 6
SECTION 2.2 INTERMITTENT PROGRAMING MODEL 7
SECTION 2.3 COMPILER AND TRANSLATOR 8

CHAPTER 3. APPROACH AND IMPLEMENTATION **10**

SECTION 3.1 ENERGY INSPECTOR 10
SECTION 3.2 TASK SETTER 12
SECTION 3.3 RECONFIGURABLE CONTROLLER 14

CHAPTER 4. DESIGN SPACE OF INTERMITTENCE **16**

SECTION 4.1 DERIVATIVES OF AMBIENT POWER 16
SECTION 4.2 SOFTWARE DESIGN SPACE 18

CHAPTER 5. BENCHMARKS AND METHODOLOGY **19**

SECTION 5.1 HARDWARE BACKGROUND 19
SECTION 5.2 SOFTWARE BENCHMARKS 19
SECTION 5.3 METHODOLOGY 20

CHAPTER 6. EVALUATION **21**

SECTION 6.1 VALIDITY 21
SECTION 6.2 COMPILATION PERFORMANCE 22
SECTION 6.3 ENGINEER EFFORT 24

CHAPTER 7. LIMITATIONS AND CHALLENGES **25**

SECTION 7.1 COMPILATION FOR INTERMITTENCE 25
SECTION 7.2 HARDWARE 26
SECTION 7.3 EVALUATION CHALLENGE 28

CHAPTER 8. CONCLUSION **29**

REFERENCES **32**

ABSTRACT

Intermittent computing describes the computation of computer systems that suffers from intermittently-powered energy source. As technologies of energy-harvesting and ultra low-power systems accelerate, diverse hardware types for intermittent systems increasingly emerges, meeting the growing demands of low-power embed systems and chips, which draws attention on the software design space of intermittent programs. Task-based, channel-based and checkpoints are ones of the prevalent solutions for intermittent programming. Task-based programing allows a long program to be intermittently executed on low-power devices. However, manual task decomposition is laborious and risky to create non-terminating tasks because no reference on execution energy cost. Non-terminating tasks refer to the tasks that exceed the energy capacity of a low-power device and consequently never terminate. Every time the device reboots, the non-terminating task will start over in a loop.

This paper aims to automate the processes of manual task decomposition and energy cost evaluation in task-based programming. Thus, we propose Sloth, a compiler to dynamically estimate energy consumption and automatically define task granularity, which is composed of three modules: energy inspector, task setter and reconfigurable controller. Sloth decomposes a program into efficient tasks based on the input of hardware energy capacity and execution energy cost. The method of energy cost estimation in Sloth is reconfigurable and can be replaced by other static estimation methods, while the syntax of task boundary is customizable to adapt different implementations of task-based programs.

Five programs styled in functional programming are used to validate Sloth. Sloth decomposes competent tasks based on energy cost and capacity. The generated tasks-based programs are successfully run on a battery-less programmable sensor device Moo. The experiment shows Sloth is able to automate task decomposition for volatile and non-volatile memory, and avoid non-terminating tasks efficiently. The performance of Sloth depends on the complexity of the input program.

CHAPTER 1. INTRODUCTION

Rapid changes in computation and information technology bring rapid changes in programming models and development tool chain. The acceleration of highly efficient processors and low-power embed systems makes it possible for computing systems to function through energy harvested from the environment, such as solar energy [8] and radio waves [26]. Table 1 lists multiple ambient power collection sources and applications. In the current market of intelligent products, battery greatly imposes the product size and service life, which is a contemporary bottleneck of computing systems. The development of battery-less systems drives potential applications in the field of ubiquitous computing (UC) and Internet of Things (IoT). The biggest reason why energy harvesting technology has not been widely used is that the energy collected by its energy collecting end and the energy consumption end is not proportionate. There has always been an imbalance between these two. With the continuous reduction of power consumption of sensors, MCUs, RF and other devices, and the breakthrough of micro energy harvesting power management IC technology, the system energy balance is gradually harmonious.

Table 1. Energy harvesting technologies

Type	Source	Application
Luminous energy	Solar energy, electric light, etc.	Solar battery, Solar semiconductor converting light into electricity
Thermal energy	Solar heat, waste heat, body temperature, etc.	Thermoelectricity
Mechanical energy	Vibration, step pressure, etc.	Electromagnetic induction, piezoelectric effect, etc.
Electromagnetic waves	Radio waves, wireless LAN, etc.	Rectenna

Ambient power and intermittent computing technology can be used to collect ubiquitous energy in our daily lives and utilize the power to construct very interesting applications, such as water management system, smart bridges, battery-free remote control, self-positioning system and a key-less door, etc. In daily life, if you use a battery for these applications, you need to manually replace the battery. Batteries are cheap, but in many cases it is often expensive to replace the batteries in the infrastructure, and the used batteries are contaminants. At this time, the energy collection scheme is the solution. For battery-less applications of energy harvesting, one of the important areas of interest is the application of the sensor network in the Internet of Things, the sensor network, and the use of sensor nodes in such systems. The RF module of the communication realizes semi-permanent power supply. In addition to the above-mentioned sensor network, software design space in programming models adapted to the intermittent execution environment are of great significance in the field of ultra low-power technology.

The prevalence of these intermittent systems urges solutions for intermittent programming. Energy harvesting and ultra low-power systems suffer from energy depletion all the way. If the program progress can not be consistently obtained, it will start from the beginning every time the system reboots, which will lead to a non-terminating execution. In order to execute long-term programs correctly on intermittent systems with limited power supply, task-based programing has been introduced to ensure continuous execution progress [6, 17]. Intermittent programs are divided into sequential code snippets called tasks to avoid never ending loops. The size of task depends on the energy capacity of the hardware and should guarantee the energy cost is safely less than the energy capacity. With task-based model, long-term programs can be safely executed on intermittent devices. However, most of these programing models require manual workload on task decomposition, which means engineers have to evaluate the energy cost of code statements over their experience. This will cause inefficient tasks division: oversize or overcautiously small. An oversize task will be non-terminating and an overcautiously small size will trigger excessive task transition actions, such as storing progress to non-volatile memory and restoring task-shared variables from non-volatile memory. Therefore, we propose Sloth, which is a reconfigurable compiler that adds efficient task boundaries to steady programs based on the hardware power cycles. The methods of code power consumption estimation and boundary placement are customizable. In this way, Sloth empowers engineers to select existing power estimation methods or use their own algorisms, and is compatible to different task boundary grammars from different task-based solutions.

We developed an implementation of Sloth as a derivative of C extension. Sloth consists of two executors: energy inspector and task setter, and a reconfigurable controller. Energy inspector evaluate the power consumption of the execution and generate a mapping of code statement and the corresponding power cost. Task setter compares the power cost of each unit of code statements with the hardware power amount its capacitor can buffer and then set boundaries for each task. The granularity of each task unit is based on the maximum sum of power consumption of code statements that less than the hardware power capacity. Sloth is reconfigurable by replacing the acquiescent energy inspector and task setter with customized C extensions or binary executable programs. With the energy inspector of Sloth, engineers are able to analyze the power consumption of their programs and skip the laborious workload of task decomposition and the disadvantage of power misestimate, causing a forward progress problem of task-based intermittent programs. These tasks will never be terminated and congest all the following tasks because the hardware energy capacity is not enough to finish the task execution. The same task will be started again after a power break cycle. Sloth does not support other programing languages other than C, but we believe it can be extended to different languages with minor efforts.

Different from static energy evaluations [7], power cost estimation in Sloth is layered on Powertop [1]. Powertop is a Linux tool for diagnosing power consumption and power management issues. It was developed by Intel to enable various power save modes in the kernel, user space, and hardware. On the other hand, boundary placement in Sloth adopts task syntax of Alpaca [4] by default. Engineers can replace it by editing a configurable document, or replace the whole function by a customizable configuration file. The reconfigurable feature of Sloth gives engineers the power to adapt the compiler to their own demands. For example, a preferable static power evaluation method such as [7, 15] can be used to replace the execution energy estimate module of Sloth. After power estimation, a mapping data set will be created to store the power consumption of each code unit. Sloth compares the power consumption of the first code unit with the hardware energy capacity. If the hardware energy is greater than the power cost of the code unit, the power cost of the next code unit will be calculated together, until the sum of these code units surpasses the hardware energy capacity. The combination of code units whose energy sum exactly right less than the hardware power capacity will be categorized as a task. The granularity of a code unit can be customized and is one code statement by default. The task setter of Sloth then set boundaries to the categorized tasks. Acquiescently Sloth uses the boundary style of Alpaca [17], such as `transition_to([task_name])` and `task`. Likewise, task setter can be customized by C extensions or working binary programs, as long as the input and output formats keep compatible with those of Sloth task setter.

We evaluate the implementation of Sloth in the aspects of validity, compilation performance and engineer effort. We use five functional programs written in C language, decomposing them into executable task-based intermittent programs, to verify the correctness of Sloth generation. The five generated programs accord with Alpaca syntax. We run them independently with Moo [29], a programmable energy harvesting hardware for testing of intermittent computing. Moo collects RFID (Radio-Frequency Identification) reader energy through UHF (Ultra High Frequency) band, engages with the RFID reader, and copes with its integrated accelerometer and thermal sensors. Each of the five programs was run twice and they were successfully executed on Moo and gives the correct results. Timestamp logs are set at critical nodes of Sloth to track necessary time period for performance evaluation. Average compilation time is 17.2 seconds, which also depends on how long and how complex the program is. Generally, the "greater" value the time complexity is or the greater the code amount is, the longer compilation time Sloth requires. We compare Sloth with state-of-the-art runtime compilers on engineer efforts. Sloth remove the manual workload of task decomposition and task connection of control flow and transition statements, replace the manual gross estimation of task energy dissipation with an automated energy comparison between expense and buffer capacity.

Additionally, intermittent programs bring possibilities to ubiquitous computing with the network of extremely low-power devices. Without the boundary of battery, energy collecting technology will play an important role in the pervasive system network. The robustness of intermittent programs gives ubiquitous computing a better software sustainability and a strong infrastructure. Developer tool chains like CleanCut [7] and Sloth impulse the diversity of engineer communities. With more efficient developer tool chains and integrated development environments (IDE) for intermittent computing and ultra low-power devices, the applications of ambient power are getting more flourish. For example, pervasive activity recognition system, ubiquitous data storage and minor temperature control systems can be better implemented through the battery-less devices. These kinds of applications require reliable software design space, a robust intermittent environment, as well as an effortless development environment. Sloth to a certain extent contributes to the development ecosystem of intermittent computing, based on multiple outstanding prior works, including ultra low-power systems, intermittent programming models and the specialized developer tool chains, compilers and translators. Many microcontrollers have a variety of low power and sleep modes that reduce product power consumption and extend battery life. Because of the differences between each type of microcontroller, developers need to understand the corresponding devices, specific analysis of specific issues. Right programming models and developer tool chains gives indication on the right hardware specs.

In the following chapters, we discuss related work from the dimensions of ultra low-power hardware, intermittent programming model and developer tool chain in Chapter 2. We introduce and compare different implementation on intermittent programs and compilers. After that, we propose Sloth, the reconfigurable compiler for task-based programming and present the realization principle of its implementation in Chapter 3. Then we discuss the design space and applications of intermittent computing in Chapter 4, as the background of the benchmark and methodology that we describe in Chapter 5. With the design space and benchmark introduced, we interpret the evaluation of Sloth in the aspects of validity, compilation performance and engineer effort in Chapter 6. After that, we present our thoughts about the limitations of Sloth and the possible challenges in intermittent computing in Chapter 7. Finally, we make conclusions and speculate future possibilities in Chapter 8.

CHAPTER 2. RELATED WORK

As technologies of embedded systems accelerate, a greater number of advanced low-power hardware or energy-harvesting devices emerge [20, 29], giving rise to numerical programing models and software implementations for intermittent computing [6, 16, 17, 27]. To efficiently utilize the programing models, developer tools such as compilers and translators also increasingly appear [7]. In this chapter, we introduce related work in the following aspects: extremely low-power hardware such as energy-harvesting devices, existing programming models for intermittent computing and developer instruments such as compilers and translators for higher level programming languages. Sloth is affiliated to the category of compiler and translator.

Section 2.1 ULTRA LOW-POWER HARDWARE

The prevalence of extreme low-power devices follows the cumulatively emergence of energy-harvesting devices, devices that collect energy from their environments. These devices harvest power from radio waves, solar energy, thermal energy and oscillation, etc.

Ultra low-power hardware stores energy in a capacitor to a threshold amount before booting the system. After starting the system and running certain tasks, the energy will soon be depleted. The system will continuously experience the loop of booting up and shutting down. The time gap a program can use between power on and off is narrow. Only a small number of code statements can be executed each time. The effective period ties to the capacitor size. With the more power it can buffer, the more statements the device can execute in each cycle. Representative energy-harvesting device such as WISP could stimulate the hardware on and off for hundreds of times per second. Volatile state of an intermittent device involves memory state, register file and execution progress, etc. Data reserved in volatile memory such as variables and intermediate results, are lost after a power-off while those in non-volatile memory persist. Execution progress is stored in non-volatile memory to keep it consistent.

The programmable WISP (Wireless Identification and Sensing Platform) [20] and the battery-less system Moo [29] are commonly-used energy-harvesting platforms for intermittent computing researchers. The energy extracted from the environment is usually extremely scarce and not sufficient to supply the hardware for finishing expected functions. This problem calls for solutions from the software design space. Intermittent programing models rise in response to this software demands.

Ultra low-power hardware and energy scavenging devices have been applied to different domains in recent years. The characteristics of battery-less and small-size bestow it the ability to embed into omnipresent environments. The pervasive sensors and computing devices give possibilities to the realization of ubiquitous applications, such as automatic speech recognition (ASR) [28], human body motion detection [5] and body-worn sensors [19]. The derivatives of ambient power impulse the exploitation of programming model and development toolkit for intermittent computing.

Section 2.2 INTERMITTENT PROGRAMING MODEL

The disorder of intermittent computing is problematic due to its possibility of producing unexpected behavior through the right program that performs normally with a continuous power supply. Program progress forwarding is one of the typical challenges of intermittent computing. For task-based programs, if the power dissipation of a certain task is greater than the energy capacity, the task will never be terminated and be repeated in a power cycle. Thus, the task granularity and the threshold value of a low-power capacitor should be defined properly. Data consistency is another critical issue of intermittent programs. Non-volatile or volatile memory require different strategy for data consistency. Volatile memory has a great performance but it loses data values instantly after a power break. Non-volatile memory stores permanent data at the expense of reading and writing speeds. In allusion to either kind of volatile and non-volatile memory, different programming models are proposed. For example, DINO [6] and Mementos [16] are an execution model undertaking volatile and non-volatile data consistency while Ratchet [27] aims at high- performance non-volatile memory.

In reply to the software demands for the energy shortage suffered by the ultra low-power devices, multiple intermittent programing models [6, 12, 16, 17, 27] are proposed, playing an indispensable role in the filed of intermittent computing. Chain [6] is a task and channel based programing model with check-pointing assuring progress and variables consistent, while Alpaca [17] remove the need of check-pointing with a data privatization strategy. In order to keep a correct forward progress, task-based model is generally adopted by most intermittent programs. The identification of the last executed task is stored as progress data, such as current version, in non-volatile memory. Getting the task executing progress will be the first thing to do after a boot-up. Only if the previous task is successfully executed, the following task will get to execution. Energy management should be tackled to avoid non-terminating task.

There are two type of power analysis and optimization methods, dynamic and static inspection. Dynamic analysis measures power consumption over an execution while static inspection is applied to a static program, no need of execution. CleanCut [7] proposes a non-terminating checker with a static power analysis over code statements. Static analysis does not require an execution and returns analysis results on the phrase of compilation. On the other hand, simulation to an actual hardware environment is usually required for dynamic inspection. Execution power consumptions of two equal programs are unequal and depends on a specific hardware design. Sloth uses Powertop to implement dynamic power analysis. We believe dynamic power analysis is reliable to actual hardware environment, sparing less processor or sensor power differences. Energy Inspector (Section 3.1) of Sloth analyzes and estimates power cost based on program execution.

Section 2.3 COMPILER AND TRANSLATOR

Higher level of intermittent programing languages or models increasingly emerge with the development of intermittent hardware, which urges for solution on compilers and translators for intermittent programs, aiming to incubate a developer-friendly engineering community of intermittent computing. Alpaca [17] is a typical task-based programing model made of C extension, which requires manual task decomposition and energy management. Meanwhile, Alpaca is also a compiler for automating the work of writing privatization oriented statements for scalar and array variables, and the two-phrase commitment. The emergence of compilers and translators energize and diversify the development of intermittent computing.

Moreover, criteria such as type of electronic products, battery capacity and power consumption are essential to hardware development for long. Chip manufacturers dare to propagandize the tag of low power consumption on the first page of their product specs. But how to define low power consumption? Leaving the hardware application, statistical value seems to lose its primitive meaning. In short, it is the smaller the better. Thus, in addition to reasonable programming models for intermittent computing, the quality and performance of the intermittent programs is also vital. An intermittent programmer should minimize the power cost of the program during development while keeping it idempotent. A specific application usually consists of many sub-functions and sub-tasks. An application of embed systems is implemented by calls to different services including hardware services, such as AD (Analog-to-Digital) sampling, serial communication, external interrupt triggering and timer services, etc., or software services, such as various communication protocol stacks, FAT (File Allocation Table) file systems, queues, software filtering, etc. A service usually implements one or more functions (a well-designed task partitioning does not allow a service to contain more than 2 unrelated functions). Simple functions, such as the CRC (Cyclic Redundancy Check) function, can get results immediately after entering the function. However, tasks are usually not the best implementation of complex functions because tasks may involve operating systems, schedulers, or state machines. Generally, the implementation of a task can be understood as a process. Since it is a process, the task is to be cyclical. For example, an AD sampling task consists of at least three steps: channel selection and start sampling, sampling and waiting for sampling completion, and data processing. These three steps together form a task cycle. When the three steps are completed, we can conclude that one cycle is over. The complexity of sensors, task functions and execution cycles is intricate and demanding simplification. Compilers, translators and testing tools are of great importance consequently, to keep the intermittent programs idempotent and robust.

For example, task-based compilers statistically determine if the power consumption of a task surpasses the maximum device capacity, which will lead to a never completed task. With this kind of compilers, non-terminating tasks will be automatically divided, or instruct the engineer to decompose the task. Sloth is a reconfigurable compiler for task-based intermittent programming models. It improves the efficiency of task decomposition and optimize the utilization rate of intermittent hardware by replacing the experience based energy estimation with dynamic analysis on power consumption. Different from Sloth, CleanCut [7] uses static energy estimation on the energy measurement. CleanCut reports failure when there is no proper placement existing, a replacement that eschews non-terminating tasks on the basis of its energy measurement method. On the contrary, Sloth separates programs into minimum code statements and combines them until the energy sum exceeds the hardware energy capacity, which means there is no task setting failure in Sloth unless the power consumption of a single basic code statement (not a function call) already surpasses the energy capacity. On the other side, Sloth supports reconfigurable task boundary setter and customizable energy consumption method.

CHAPTER 3. APPROACH AND IMPLEMENTATION

In this chapter we introduce Sloth and its implementation. We develop Sloth as a compiler of intermittent computing, so as to replace the the process of manual task decomposition with automated energy cost based task generation. Sloth is a reconfigurable compiler and we design it as module by module. With a modular infrastructure, Sloth consists of three modules: (1) energy inspector, (2) task setter and (3) reconfigurable controller. We will interpret each of these three modules in the following sections. We aim to develop a low overhead, reusable and reliable runtime library of Sloth with a proper performance. At present, the implementation of Sloth supports programs written in C and in the environment of Linux. Ambient power devices execute intermittently and experience energy depletion accidentally and repeatedly. Desultory execution impedes progress and results in discordant data and variable. It is not intuitive and requires extracting a new development pattern. Sloth tends to ease the demand and gives possibilities to intermittent developers by providing a more intact developer-friendly tool chains.

Section 3.1 ENERGY INSPECTOR

To properly decompose a program, a statistical criterion of decomposition should be obtained. We need to check the energy cost of each unit of the whole program and define the size of each unit. If we have the hardware energy capacity and the energy cost of each code unit, we can compare the sum of energy cost of multiple units with the threshold power capacity and take the maximum number (size) of units that does not exceed the power capacity. With this criterion, Sloth will decompose the program into efficient tasks for the targeted task-based model. To achieve this, we divided the program into the smallest unit based on code statement: one statement as one unit. Sloth separates the program into multiple code statements and stores them in the first column of a CSV (Comma-separated values) file. The second column of this file will be filled with energy consumption corresponding to the code statement of the first column. This CSV data will be used as a measurement for the task setter (in Section 3.2). With regard to code blocks such as `"If, While, For"`, code statements inside will also be inspected one by one. The cycle index of `"While, For"` loop will be calculated and reflected at the number of the same code statements in the CSV file (As shown in Algorithm 1). For example, a while loop with Statement A and Statement B will repeat for three times, and then in the CSV file, there will be six statements for this while loop, in the sequence of "A, B, A, B, A, B". However, there are occasions when the cycle index can not be predicted, such as the argument of repeat times is an input to the program, which is defined by the caller. In this case, the energy cost of a single loop will be calculated and a warning will be output to the developer, notifying this uncertainty of the program. This uncertainty has the potential to cause forward progress issue if it is not properly handled. The task containing this uncertainty might never finish execution, even if it has been running for many times, because its energy cost surpasses the energy threshold.

In Algorithm 1, function `CodeBlockGetter` gives the list of characteristics of `"If, While, For"` code blocks for future usage. Then, function `StatementReader` return the separated units by the granularity of single code statement and code blocks. If it encounters a code block, it will categorize it as one temporary unit. After `SlothEnergyInspector` comes into contact with this kind of temporary code block unit, it iterates itself with this code block unit as the argument and continue to the next code unit after processing this temporary code block with the same logic. If the code unit is a code statement, Sloth arouses Powertop and calculate the power energy of this code unit. `PowerttopEstimate` function makes this code unit to an executable program with all previous direct or intermediate variables and runs it. Sloth will store all the direct and intermediate variables of the program. After that, Powertop is called and Sloth gets the energy cost of this provisional program. If the energy cost of a smallest code unit exceeds hardware energy capacity, a warning will be alerted to the engineers. Otherwise, multiple adjacent code units will be combined until the sum of their energy cost surpass the hardware energy capacity. The threshold value of the hardware energy capacity that we used is the minimum energy capacity excluding the buffer of possible energy lost.

Algorithm 1. Pseudo-code for Sloth Energy Inspector.

```
function SlothEnergyInspector (Program P)
    codeBlockList ← CodeBlockGetter()
    statement ← StatementReader(P)

    for statementUnit ∈ P do
        if statement ∈ codeBlockList then
            SlothEnergyInspector(statement)
        else
            energyCost ← PowertopEstimate(statement)
            storeResult ← StoreEnergyCostToCSV(statement, energyCost)

    return storeResult
```

Specifically, in function `PowertopEstimate` (As shown in Algorithm 2), Sloth get the direct or intermediate variables from storage, which is constructed at the beginning of the Sloth compilation. Sloth stores all the direct assigned variables and then run the whole program with continuous power. During the execution, Sloth reserves the intermediate variables in the order of code index. For example, if the execution runs to line 5 of the program, all the direct assigned and intermediate variables (`variablesSoFar`), which are computed over direct assigned variables of scalar or structure data types, are reserved to the key of line 5. When the function `PowertopEstimate` handles the statement with the key of line 5, the direct and intermediate variables to line 5 will be read and used to create the provisional program. This program is the carrier of the corresponding code unit and its power consumption is estimated by Powertop.

Algorithm 2. Pseudo-code for Sloth Powertop Estimate.

```
function PowertopEstimate (Statement S)

    variablesSoFar ← GetVariablesSoFar(S)
    provisionalProg ← CreateProvisionalProg(variablesSoFar, S)
    runResult ← runProvisionalProg(provisionalProg)

    if runResult equals True then
        energyCost ← getPowertopResult()

    return EnergyCost || False
```

Generally, Sloth executes every code fragments with Powertop after separating the program into minimum units. Powertop calculates each code units' energy consumption and record them as mapping data. Sloth then categorize the code statements into different combinations with the power capacity as combination granularity. These combinations will be future tasks after being processed by the task setter. For `If` blocks, since code statements will only be executed once, statements will be treated as normal statements. For code statements in `"While, For"` loop, traversal number will be calculated. Whether to separate the loop into different code combinations depends on how much energy this loop consumes. Only safe statement number not leading to non-terminating progress will be categorized into a task. For example, if a `While` loop is composed of 10 iterations, each iteration consumes one energy unit and the minimum hardware power capacity is two energy units, this loop should be decomposed into 5 individual runtimes with the same task content.

Section 3.2 TASK SETTER

After Sloth acquires the criteria of energy consumption to categorize different code units through the energy inspector (Section 3.1), we need to actually construct the tasks over these code units and their power consumptions. Task setter of Sloth compares sum of energy consumption of sequential code units with the hardware power capacity, the threshold value of energy buffer, to get the maximum number of sequential code units that can be categorized as one task. In the back of these comparisons, task setter inserts task boundaries to specific code combinations using the syntax of Alpaca by default. Sloth task setter makes recursive calls until all the code units have been handled. If the energy consumption of a single code unit surpasses the energy that the hardware capacitor can buffer, a warning will be output to notify the developer. This code statement should be simplified or optimized, or change a hardware device with a greater energy buffer. Otherwise the program will be non-terminating to the intermittent environment provided by this original hardware.

Algorithm 3. Pseudo-code for Sloth Task Setter.

```
function SlothTaskSetter (EnergyCapacity C, CodeUnitIndex I, Task T)

    codeUnit ← GetCodeUnit(I)

    if codeUnit is Empty then
        if T is not Empty then
            SetTaskBoundary(T)
            T ← ClearTempTask(T)
        return False                  // Termination

    attemptedPowerSum = T.powerSum + codeUnit.powerCost

    if attemptedPowerSum <= C then
        PushToTask(T, CodeUnit)
        T.powerSum = attemptedPowerSum
        SlothTaskSetter(C, I + 1, T)  // Recursion
    else
        if T is Empty then
            warning('power cost of unit %d exceeds capacity', I)
        else
            SetTaskBoundary(T)
            T ← ClearTempTask(T)
            SlothTaskSetter(C, I, T)  // Recursion
```

In Algorithm 3, Sloth task setter get the object of current handling code unit or statement through the function `GetCodeUnit` at the beginning of the execution, from the CSV file generated by the energy inspector (Section 3.1). If there is no corresponding code unit according to the index, an empty object will be returned. In this case, there are two possibilities: wrong index is input or the iteration reaches the end of the code unit list. Task setter terminates in either scenario. If the temporary task variable `T` is not empty, Sloth will wrap this task up and end this execution, in case this is the last iteration of the program. If Sloth get code unit successfully by the index argument, task setter will accumulate the power cost of the current unit to the power consumption of the temporary task `T`. If this attempted power sum (`attemptedPowerSum`) is less than the hardware energy capacity, this code unit will be added to the temporary task and Sloth make a recursive call to the next code unit. Alternatively, if the attempted power sum is greater than the hardware power capacity, two cases will be considered. Firstly, if `T` is empty, which means the power cost of this single code unit already exceeds the hardware power capacity, a warning will be pushed to the developer, indicating to optimize this code unit or change the hardware to another one with higher power capacity. Secondly, if `T` is not empty, that means the power sum of the previous code units plus that of the current code unit exceeds the threshold value of power buffer, the current code unit should be excluded and the previous code units should be categorized as one task. Sloth then combines the code units that the temporary task `T` contains into a task and set boundary to this task. After this, Sloth clears `T` and makes the recursive call to process the same logic to the current code unit again. The whole process traverses from the first code units that are separated through the energy inspector to the last one of these code units.

With all these processes Sloth decompose an intact program into efficient tasks that adapt to the appointed hardware specs. Energy inspector of Sloth separates the program into smallest code units by default, which is separating one code statement as one code unit. With the smallest granularity of the code units, the sum of the smallest power consumption is attempted. In this way Sloth avoids coarse task combinations and uses the best efficiency of the hardware power, maximizing the performance of the program. With energy inspector and task setter, Sloth replaces the manual workload of task decomposition and power management with efficient execution-ready automation.

Section 3.3 RECONFIGURABLE CONTROLLER

Energy inspector and task setter guarantees the task decomposing automation to be safe and efficient, but they only stick to one task-based programing model, Alpaca. However, there are multiple task-based intermittent programing models, such as Chain, Mementos and DINO, etc. As task-based models are prevalent in intermittent computing and increasing derivatives are coming up, compatibility to only one model is far from enough. Thus, to adapt Sloth to different task-based programing scenarios, we add the module of reconfigurable controller to it. Reconfigurable controller provides a modular structure to Sloth and makes energy inspector and task setter two independent modules. The modular and hierarchical structure of Sloth also gives the compiler good scalability, maintainability and portability. It is easy to prevent errors, and improve program stability in this way. According to the characteristics of the C language, we organize the program using a hierarchical and modular software development model. Each module can only use the interface provided by the layer and the next layer of modules. Each module's file package exists in a separate folder. Usually, there is more than one file to implement a module, these related files are saved in a directory. The conditional compilation macro for module reduction is saved in a separate file for software reduction. The declaration and definition are separate, use the `*.h` file to expose the module needs to be provided to external functions, macros, types, constants, global variables, try to make the module transparent to the outside, the user does not need to understand the specific implementation when using the module function.

The reconfigurable controller is divided into different modules according to the function. The main function only calls the function, not defining the function. Definition function in each module file uses the function with the same name to declare the external function for other files to call. We give instructions on how to replace the energy inspector or task setter with the same input and output formats to developers. Developers write their own inspector and setter into different files and then pass the file sources to Sloth. Sloth will try to reach these files and use them to inspect energies and define tasks. For example:

```
$ ./Sloth --inspector ~/customized_inspector_file --setter ~/customized_setter_source
```

For Sloth, at compilation time, the compiler needs to check whether the syntax is correct, whether the declaration of functions and variables is correct; only the declaration of functions and variables without definition is correct. The function declaration tells the compiler that the function already exists, but the entry address has not yet been determined. When linking, the compiler will find the function entry address and replace the tag. After these are verified, the compiler can compile the intermediate object file. In general, compilation is for a single source file, multiple source files need to be compiled multiple times, and each source file will generate a corresponding target file. The purpose of the link is to find the function entry address and organize all the source files into a binary file that can be executed. We put the declaration of functions and variables into the header file, and then `#include` in the current source file, the global variable is declared as static, only visible in the current file and not exposed to external usage and we use macro definition to external usage. We replaced the global variable with a macro definition, and placed both the function declaration and the macro definition in the header file `module.h`, so that only `module.h` needs to be included in `main.c`. Both the `.c` file and the `.h` file are source files. Except for the suffixes that are different for distinguishing between external and management, the others are almost identical. The code written in `.c` can also be written in `.h`, such as function definition and preprocessing. However, the `.h` file and the `.c` file assume different roles in the project: the `.c` file is primarily responsible for implementation, that is, defining functions; the `.h` file is primarily responsible for declarations, such as function declarations, macro definitions, and so on.

CHAPTER 4. DESIGN SPACE OF INTERMITTENCE

Collecting energy sources such as heat, light, vibration, and electric waves, and then converting them into electricity to realize battery-free equipment is an energy-harvesting (EH) technology called environmental power generation. Environmental power generation arouses software design space solutions due to the conundrum of intermittent energy supply.

Section 4.1 DERIVATIVES OF AMBIENT POWER

As one of the typical derivatives of ultra low-power hardware, wearable devices have developed various types of watches, wristbands, and glasses, and it is said that they will enter the real era of universalization. Although it is also known as the ICT (Information and Communications Technologies) device after smart phone, it is not a smart phone, but a small terminal connected with a smart computer. It can also be used to operate computers and external devices, such as smart televisions. In addition, by installing various sensors, it is possible to wirelessly transmit vital signs such as heart rate and blood pressure to a smartphone, thereby contributing to health management and the like, and its application in health care and medical fields is also expanding.

EH technology is highly compatible with wearable devices and wireless sensor networks (WSNs) and is expected to be used in a variety of applications [25]. The interconnection with the smartphone requires the use of wireless communication technology, and BLE (Bluetooth Low Energy) is proposed in order to improve the battery life. BLE is a low-power communication specification added to Bluetooth 4.0 [9]. The device that supports BLE is "Bluetooth Smart", and devices such as smartphones and tablets that become hubs are called Bluetooth-Smart-Ready. In addition, wearable devices such as ring-type and pendant-type jewelry types have also been developed, and it is said that in the future, "wireless communication tags and sensors will be mounted on items such as daily necessities, kitchen supplies, and stationery, and other IoT (Internet of Things) applications". However, this has also created a huge technical problem, that is, the battery problem. Because even wearable devices that use BLE that support ultra-power saving are currently required to be recharged frequently.

NB-IoT (Narrowband Internet of Things) is mainly used in IOT devices, individuals, utilities and industry, such as Internet of Things devices, personal applications, public utilities and industrial applications. IoT devices involving low-power wide-area IoT provide users with real-time information transmission through sensors and wireless communication modules. NB-IoT's ultra-small chip can make wearable products more compact, and the deep coverage of the network can make small antennas also achieve good results. In addition, the requirements for more application scenarios such as positioning and tracking are also very clear. The global public utilities industry has initiated or plans to initiate the transformation of utility applications such as smart meter reading, smart trash cans, smart environmental protection, and smart parking. The narrowband Internet of Things is opening up a vast market that has never been seen before. The application of narrow-band IoT technology in industrial manufacturing will bring about industrial manufacturing transformation and upgrading. Nowadays, the factory is gradually applying wireless connection technology to products or production lines to improve the manufacturing process. At present, the wireless connection technology like Wi-Fi is not safe enough and the reliability is not high enough. NB-IoT is ideal for industrial IoT wireless connection. solution.

Energy harvesting is both old and new. The ore radio used in the early 20th century is a battery-less radio receiver that uses the energy of the received waves to make the headphones sound. In addition, solar calculators and solar watches using thin film solar cells are also energy harvesting devices. In recent years, energy collection technology has once again attracted attention because of the development of ICs with low power consumption, etc., thereby increasing the possibility of implementing auxiliary power supplies for mobile devices and sensor modules without batteries.

Unlike renewable energy such as solar/wind power generation, the amount of power generated from energy collection is only µW (10-6W) to mW (10-3W), but if it is such a level of energy, we will be around. In addition, through the combination of sensing technology and wireless technology, it is possible to construct a sensor network system that does not require exchange and charging of batteries. For example, if a human sensor is used, automatic control of lighting and air conditioning can also be performed, which will become an important technology supporting intelligent building and intelligent residential energy management systems. Moreover, it is also expected to be applied to sensing in a harsh environment where power supply or human inaccessibility cannot be ensured.

In addition, a research on ambient power presents a battery free cellphone that obtains power from sunlight or RF (Radio Frequency) waves emitted by nearby communication base stations [24]. With backscattering technology, the phone can make a voice call by changing and reflecting the same wave to the base station. The current model of this mobile phone model consumes only 3 microwatts (uW), which is one-tenth of the current power consumption of smartphones. The phone is available for voice and emergency calls. This technology may be more suitable for developing countries because there are not many reliable sources to charge mobile phones. And because the materials used in mobile phones are readily available materials, they are much cheaper than the current smartphones on the market, which means more people can afford it.

Section 4.2 SOFTWARE DESIGN SPACE

To tackle the intermittent computation caused by the intermittent power supply, thereupon it comes the intermittent programming models. Task-based programming models are presented and becoming the most prevalent models with numerous derivatives, such as Alpaca, Chain, DINO and Mementos, which are introduced in Chapter 2. In addition to making robust intermittent programming, energy-efficient design for application compilation is likewise vital to ultra low-power devices. Reducing redundant code is the first concern. When the processor processes the system, it consumes 30% of the energy consumed by the cache. In addition, if the cache cannot be hit, the content is exchanged. Therefore, the external bus is driven to increase the energy, and the redundant code is appropriately reduced when the program is compiled. This will greatly reduce the cache activity, in order to achieve the purpose of reducing system power consumption. Secondly, intermittent programs urge optimization on I/O power consumption. In the specific gravity of the system, the energy required to drive the I/O port has a large proportion in the whole system. Therefore, in order to reduce the system energy consumption, the number of application-driven I/O ports should be appropriately reduced, that is, according to the coding design technology. We can reduce and compress I/O data to reduce I/O frequency, optimize and analyze the application to store the local access performance and the nature of the exchange bus address activity when the compiler is properly applied to achieve the purpose of compiling the code. Lastly, the system hardware needs to have a certain decoding effect. In the process of analyzing system programs, continuously optimize local high-frequency data performance, thereby reducing the frequency and frequency of operating access to the system, not only reducing system power consumption, but also providing energy for system operation.

Additionally, code compression methods are used to reduce the power consumption of the low-power embedded system. After analyzing the characteristics of the instruction, the target code is subjected to instruction merging and instruction segmentation, and the processed Huffman algorithm is used to encode the processed instruction to generate an index lookup table. Finally, the target code is compressed and decompressed by looking up the correspondence between index words and instructions in the table. The experiment uses the simple scalar simulator to compress some embedded benchmarks, and evaluates the code compression rate and power reduction rate. The statistics show that the proposed data is presented. The improved algorithm can effectively save storage space and reduce system power consumption.

CHAPTER 5. BENCHMARKS AND METHODOLOGY

This chapter reviews the background of research on low energy harvesting circuits in the environment and the research situation, and introduces precondition of benchmarks and methodology for the validation of Sloth (Chapter 6).

Section 5.1 HARDWARE BACKGROUND

With the development of the Internet of Things and the widespread use of sensors, battery-based wireless sensor power supply methods have attracted much attention due to the inherent shortcomings of batteries. By using the low energy in the environment to convert into electrical energy, self-powering to the wireless sensor network node can be realized. Researchers hope to replace the battery by collecting weak energy in the environment or use the collected energy to charge the battery to extend the life of the battery and solve the power supply problem of the wireless sensor network node. Therefore, the collection of low energy in the research environment is of great significance for the development of power supply methods for devices such as new sensors. It expounds the collection techniques of weak energy such as vibration energy, solar energy, temperature difference energy and friction energy in the environment, and analyzes vibration energy, solar energy and temperature difference energy respectively. A kind of energy harvester studied the output characteristics of these three energy collectors and compared their output characteristics.

Besides, the rise of LPWA (Low-Power Wide-Area) network has benefited from the rapid development of the Internet of Things in recent years. In order to meet the connection needs of more and more long-distance IoT devices, LPWA came into being. Low-power wide-area network designed for low-bandwidth, low-power, long-range, high-volume IoT applications. LPWA is suitable for IoT applications such as smart water meters, smart parking, asset tracking and wearable devices. Sloth and its prior and related works will have incremental value to the ubiquitous IoT technologies in the aspect of programming models and development toolkits. With the combination of LPWA network and battery-less hardware, the derivative applications in this field will be more and more diverse.

Section 5.2 SOFTWARE BENCHMARKS

In terms of task decomposition compilers, CleanCut [7] is one of the intermittent compilers that targets on task decomposition and energy measurement of intermittent programs. We compare

Sloth with CleanCut in the dimension of energy analysis and task decompose strategy. We use Alpaca [17] as the default syntax benchmark and verify the generated results of Sloth with the Alpaca syntax criteria. We use Sloth to decompose five simple C programs which originally target on devices with continuous power supply, into intermittent-ready task-based programs. These testing programs are: (1) a program to convert octal to binary [4], (2) a program to check if a number is palindrome or not [3], (3) a program to perform addition and subtraction of matrices [23], (4) a simple human body detection program [11], (5) an asterisk [2] based voice changer that transforms a person's voice pitch during a phone call in real time [10].

Section 5.3 METHODOLOGY

The current energy harvesting technology is discussed and the existing typical energy harvesting hardware is analyzed with Sloth. We use Moo [29] to run the intermittent programs that Sloth generates. Moo and WISP [20] are two research oriented intermittent devices that most intermittent computing researchers will adopt. Moo and WISP are battery-less devices harvesting RF energy. The advantages and disadvantages of several energy collection technologies are carried out. Experiments were carried out on the self-designed low energy harvesting hardware, which realized that the collected electrical energy was stored in the energy storage device or directly supplied to the load. The energy harvester is connected to the circuit to verify the feasibility of the energy harvesting circuit to store the collected electrical energy in the lithium battery, as well as the low power consumption of the circuit itself. The experimental results show that the energy harvesting hardware designed in this paper realizes the collection of vibration energy, solar energy, temperature difference energy, and power supply to wireless network sensor nodes. The power consumption of the entire wireless network sensor node is reduced to about 100μW, and the designed circuit has the characteristics of low power consumption, simple structure, low cost, etc., and achieves the design goal and has application prospects.

CHAPTER 6. EVALUATION

To verify the approach, we evaluate the implementation of Sloth in the dimension of validity, compilation performance and engineer effort. For validity, we verify the correctness of the generated task-based programs, to see whether they can be successfully executed and output the expected content. After that, we validate the compilation performance of Sloth implementation. We set log trackers in critical time point and analyze the timestamps from the log files. Finally, we discuss the engineering effort that Sloth saves for the developers.

Section 6.1 VALIDITY

We run Sloth to process five C programs (Section 5.2) which are written and tested in PC (Personal Computer) with continuous power. We verify whether the output of the processed task-based program equals to the output of the original program. In order to run the generated programs, we install the Moo (As shown in Figure 1) devices connected with a RFID Reader. Moo is a programmable battery-less hardware that harvests RFID reader energy with a 47µF capacitor. We connect Moo to the PC with the USB (Universal Serial Bus) Programmer, deploy Alpaca runtime environment on it and execute the five generated programs with it, through the interface on PC.

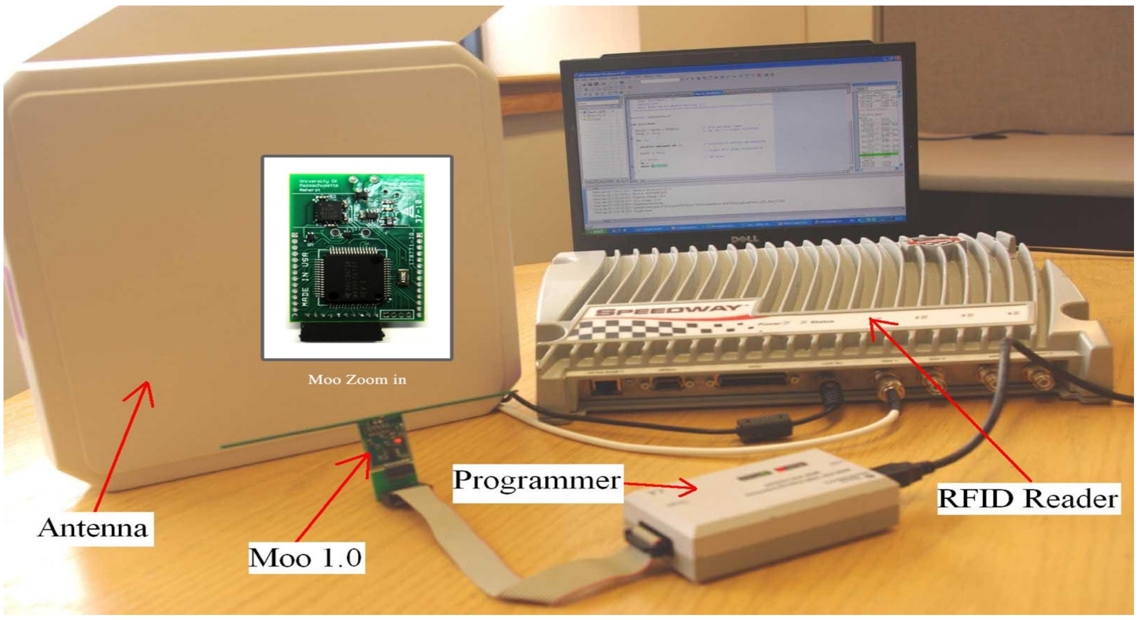

Figure 1. Moo and RFID Reader

With all the above steps we check on the correctness of Sloth compilation and its generations. All the five programs are coded with functional programming style and are within one single file. The first three programs are simple computations without complex arguments, only scalar variables. To simplify the case, when executing the task-based programs of converting octal to binary, checking palindrome number, and performing plus and minus operation to matrices, we set no input for them, only preset the arguments as variables in the programs. As to the last two programs, simple body detection and a voice changer, we prepared an image and an audio file at the same directories. These two files are the default input to these two testing program, because there is no camera or microphone in Moo to collect the information. We use Sloth to process the five programs one by one and execute the generated task-based programs with Moo. After that we compare the program results between these two type of programs in the two different environments, on PC and Moo, continuous and intermittent power supply.

Table 2. Correctness of the generated programs

The Testing Programs [1]	Program Result	Forward Progress Issue
1. Octal to Binary	Correct	No
2. Check Palindrome Number	Correct	No
3. Matrices Simple Calculation	Correct	No
4. Simple Body Detection	Correct	No
5. Simple Voice Changer	Correct	No

[1] The five testing programs mentioned in Section 5.2

From Table 2, we know all of the five testing programs behave the same, no matter in PC or Moo. All of the program results from Moo correspondingly equal to those from PC. Thus, we speculate Sloth keeps the execution validity when compiling the original program. The result shows that the generated programs are correct and the processed programs are successfully executed. No forward progress issue is detected in this experiment. It means the energy inspector of Sloth calculates the power consumption of tasks correctly. The task power consumption safely less than the power buffer of the Moo capacity.

Section 6.2 COMPILATION PERFORMANCE

Logging trackers have been set in critical program nodes to track the performance of Sloth compilation. With the logged timestamps, we analyze the compilation time of Sloth and discuss the performance of the Sloth compilation itself. Sloth compilation is a preparation phase of the final task-based intermittent programs. Thus, we don't expect Sloth to perform like user-oriented applications. However, it should be within a reasonable range. Also, the compilation time of Sloth depends on the complex of the input program. Generally, the longer the program is or the more "complex" the program is (greater time complexity), the more time Sloth needs to compile a continuous program, into an intermittent program.

Table 3. Compilation time of Sloth

The Testing Programs [1]	Average Compilation Time	Standard Deviation
1. Octal to Binary	11.91 s	0.47
2. Check Palindrome Number	14.90 s	0.38
3. Matrices Simple Calculation	18.04 s	0.57
4. Simple Body Detection	32.23 s	0.73
5. Simple Voice Changer	51.21 s	0.69

[1] The five testing programs mentioned in Section 5.2

We use Sloth to compile five times for each testing program and calculate the average compilation time of each compilation and their standard deviations. As shown in Table 3, for these five testing programs, the compilation time varies from 11 to over 50 seconds. The more complex or longer the program is, the more time Sloth needs to handle it. This is because energy inspector of Sloth traverses every code statement of the testing program, including the loops. The more code statements or more loops in the program, the more iterations Sloth needs to do, which means the longer compilation time. In the step of Sloth task setter, the compilation time of this part is basically the same among different testing programs. We believe this time quantum is acceptable to developers because usually engineers will develop the application with continuous power, like with a PC, and then decompose the program into a task-based intermittent-ready program. When developing in PC, they can easily test the functionalities of their applications. The Sloth compilation is the last step which converts program to that of intermittent environment. Developer will not experience the debugging lap caused by this compilation time because they debug in the development phase with a PC, and with Sloth, they launch it to a testing environment like Moo. Moreover, there are multiple ways to improve the speed of the compilation, such as reducing interdependence, using parallelism and shared library, etc. Moving the infrequently modified code into the library will reduce compilation time. We can also reduce the link time by using a shared library. Also, we can use a lower optimization level. The more the compiler tries to optimize, the harder it is to work, which means the longer time the compilation needs. There are always trade-offs between compilation functionalities and the compilation speed.

Section 6.3 ENGINEER EFFORT

Sloth replace the workload of developer's manual task decomposition of intermittent programs and the experience based energy estimation which might cause forward progress issue or inefficient energy usage rate. CleanCut replace similar engineer effort on task-based intermittent programming. CleanCut uses static energy measurement while Sloth adopts dynamic energy inspection during execution. These two methods serve in different dimension and fulfill the different demands in task-based energy measurement or task granularity setting. On the other hand, the reconfiguration of Sloth makes it compatible to different task-based programming models. Sloth is acquiescently compatible with Alpaca, but supports other syntax of extra task-based programming patterns. Additionally, task setter automates the code unit allocation to task and task boundary setting. With the task boundary inserted into the original program, the program become compatible with intermittent environment.

One of the most important functions of compiler is the reaction to errors in the program, which can be diagnosed at almost every stage of compilation. In the lexical analysis stage, spelling errors of characters are found. In the parsing stage, the word string is checked for violation of the structural rules of the language. In semantic analysis, the compiler further checks the grammatically correct but meaningless operational components. Sloth gives error reaction when encountering situations that Sloth can not solve by itself. For example, when the energy consumption of a single code unit is already greater than the hardware energy capacity, Sloth will terminate the compilation and output a warning to the developer, indicating the developer to decompose the code unit or optimize the code unit size before using Sloth compilation again, or even change the hardware if necessary. The code optimization phase when receiving warnings is mainly to improve the quality of the intermediate code in order to improve the speed of the target program. Code optimization can be done either after semantic analysis or after code generation. Even after a successful Sloth compilation, developers are suggested to implement program check and code optimization.

CHAPTER 7. LIMITATIONS AND CHALLENGES

When analyzing embedded systems, the most important and critical issue is the power consumption problem. To ensure effective power reduction, embedded system problems should be analyzed from a system perspective. Low-power embedded systems not only require hardware devices. It also needs the support of system software technology, reasonable optimization and cooperation with system software and hardware technology, in order to achieve the purpose of realizing the power consumption of embedded system. In this chapter, we discuss limitations and challenges of Sloth.

Section 7.1 COMPILATION FOR INTERMITTENCE

Compilation for intermittent programming to a certain extent solves the laborious workload of task decomposition. However, if a program gets complex, it takes too much time and requires a well performed computer on compilation. On the other hand, Sloth only support functional programming style, which means complex application that involves complicated and interdependent Class and Object is excluded by Sloth. Developers will have to modify their programs into a functional programming style before using Sloth. However, the programs for intermittent systems are usually simple and are supposed to be independent, so this issue should be acceptable temporarily. Moreover, Sloth can not process or evaluate the energy cost of the external devices or sensors that connected to the intermittent devices. For example, if an audio collecting sensor is attached to the intermittent device, the function of collecting audio information can not be executed on a compilation computer. Thus, Sloth can not obtain the energy consumption of the functions or APIs (Application Program Interface) of this external sensor. Developers will have to evaluate the power consumption of this sensor or ask the manufacturer for specs of this kind of information, and then add it to the energy data mapping (Section 3.1). In the experiment mentioned in Section 6.1, we modify the open source programs and reconstruct them into a single file program before using Sloth, because in this phase, Sloth does not support external files. Only the libraries that can be preinstalled are able to be included with Sloth compilation, such as asterisk in the experiment (Section 6.1). With the development of intermittent computing, Sloth are expected to keep paces with the hardware and programming model progress. The reconfigurable characteristic of Sloth can be of certain compatibility but may also have the possibility of incompatible input and output with modules of energy inspector and task setter in the future.

Section 7.2 HARDWARE

The usage and development of compilation tool chains like Sloth, CleanCut or programming models like Alpaca, DINO or Chain, are constrained by the progress of hardware approaches. If specs or performance of hardware have been greatly improved, the development of intermittent software design space will be of great changes. In order to achieve high efficiency in collecting various energy in the environment, a low power consumption weak energy collecting hardware is designed. Using the LTC3588-1 power management chip as the core voltage conversion circuit, the LTC4071 charge control chip as the core charge control circuit, and the TPL5100 as the core timer circuit to build a low-power weak energy collection circuit, the designed circuit can collect energy is converted to electrical energy stored to a lithium battery or supplied to a load. The experimental results show that the designed low-power weak energy harvesting circuit realizes the collection of energy. The average energy consumption of the energy-collecting circuit is as low as 182μW, which verifies the feasibility of collecting weak energy to power the wireless sensor network node. Power consumption, low cost and other advantages have application prospects. Thus, intermittent devices or ultra low-power hardware are meant to change from time to time, which would draw great demands on changes of compilation or programming model.

As mentioned in Figure 2, converting the kinetic energy generated by human motion into usable electrical energy to power the sensor has always been a hot spot in energy harvesting research. How to effectively use human motion, enhance environmental adaptability and improve energy harvesting performance are still key issues to be solved in captive energy research. Based on human motion characteristics, this paper designs a new hybrid energy harvester with piezoelectric and electromagnetic conversion mechanisms. Piezoelectric energy capture is based on the deformation of the piezoelectric beam to generate electrical energy. The electromagnetic generator uses a stacked magnetic group configuration to cut the coil to generate an electromotive force. Firstly, a dynamic theoretical model of the hybrid energy harvester is established to describe the output voltage characteristics and compared with the experiment. Both theoretical and experimental studies have shown that the output voltage of the hybrid emitter appears two peaks within a certain excitation frequency range. By adjusting the length of the piezoelectric beam, you can change the peak size and the range of the frequency between the two peaks. Human motion experiments show that the hybrid trap can provide higher voltage output in a short time. For example, when the running speed is 5 km/h, the 1.1 V DC voltage can be output to drive the sensor within 3 s; the running time is long. For 30 s, the sensor can often reach 77 s when it is working properly. The hybrid energy harvester designed in this paper not only can supply power quickly, but also has strong endurance, which provides potential application value for battery charging or sensor power supply.

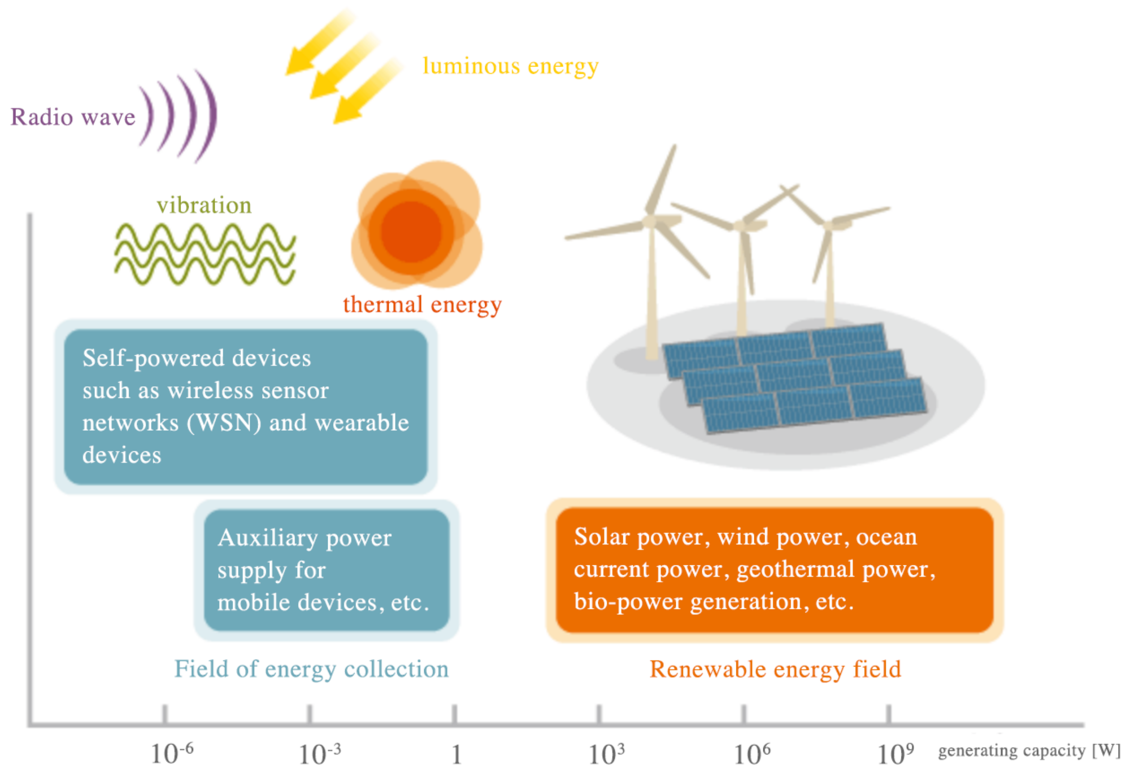
luminous energy
Radio wave
vibration
thermal energy
Self-powered devices such as wireless sensor networks (WSN) and wearable devices
Auxiliary power supply for mobile devices, etc.
Solar power, wind power, ocean current power, geothermal power, bio-power generation, etc.
Field of energy collection
Renewable energy field
10^{-6}
10^{-3}
1
10^{3}
10^{6}
10^{9}
generating capacity [W]

Figure 2. Energy generating capacity and applications of energy-harvesting

Additionally, the rapid development of the Internet of Things (IoT) is one of the key industry trends, and wireless connectivity is an important building block of IoT. One end of the IoT is the cloud, close to the data center, and the other end is the edge node, close to the sensor network and various actuators. For edge nodes, current semiconductors have a comprehensive lineup of IoT products, including sensing, interconnect, motor drive, LED (Light-Emitting Diode) lighting driver, processing / MCU (Microcontroller), wired charging, wireless charging and power management building blocks, providing a variety of packaging technologies, including multiple chip packages (such as traditional Side by Side packages, higher integration of stacked die packages), system-in-packages (integrating antennas and passive components into a single package, referred to as SiP), micro-packages, Modular packaging, etc., and through related testing and certification, help to achieve a highly integrated, energy-efficient, cost-effective innovative IoT solution.

Section 7.3 EVALUATION CHALLENGE

Evaluation of Sloth has multiple verification challenges. For example, the efficiency of task decomposition has not been verified yet, whether the granularity of the task is the best size is unknown, but we know the forward progress issue of non-termination is avoided by Sloth compilation. Theoretically the efficiency of task decomposition depends the granularity of the code units (Section 3.1). To verify the efficiency of task fragments, which means that the energy cost of a task accounts for the maximum of the energy capacity of the intermittent device. For example, if the hardware capacity is 4 energy unit, and the average energy consumption of the tasks is 3.7 energy unit, then we can say that this task composition is efficient. The actual energy cost of the tasks running on Moo is hard to measure and thus we have not verified the decomposition efficiency of tasks, which will be a future work to Sloth and a challenge on the energy measurement in the actual intermittent environment.

On the other hand, evaluation of Sloth has the limitation of excluding the execution power consumption difference among different hardware implementation. Specifically, we run Sloth and decompose the programs on Ubuntu, and thus the execution power consumption conducted by Sloth energy inspector is based on the Ubuntu environment, while the power consumption of the same task or the same code statement might be slightly different on Moo. This difference may cause inefficient task decomposition that we mentioned above, or even cause a forward progress issue and result in non-terminating programs. This possibility could be critical to an intermittent application, so we urge developers to fully test their applications before launch it to actual intermittent device after the compilation of Sloth. We are concerned by this issue but this situation of non-termination has not appeared in the experiment (Section 6.1). Future work on Sloth should focus on the execution power estimation, in addition to the automation of manual workload. When the compiler's new feature is completed, it will be preferred to run a dedicated compiler test. For big companies, this kind of test group has about 8,000 types, each of which is like 1+1 is not equal to 2, or the loop is run so many times. Many of them are verification tests. Generally, no matter how you optimize the compiler or how simple the new functionalities are, at least don't let the generated code get the wrong result.

CHAPTER 8. CONCLUSION

There are various energies in our living circle, such as light, heat, radio waves and vibrations. These energies are input into energy harvesting power management integrated circuits (PMICs) in the form of solar, electromagnetic, piezoelectric, etc. for energy harvesting. Output power can be used in sensors, low-power microcontroller units (MCUs), or in storage components such as electric double layer capacitors and all-solid-state secondary batteries. To cater the great progress of energy-harvesting hardware and programming models, we propose Sloth, a configurable compiler for program energy measurement and task-decomposition in the field of task-based programming of intermittent computing.

Sloth consists of three modules, the energy inspector, the task setter and reconfigurable controller. The Sloth energy inspector conducts power measurement to the original programs which are written in a continuous power supply. The task setter is responsible for inserting task boundaries based on the results of the comparison between hardware energy capacity and estimated energy consumption of temporary task composed of code combinations. We discuss related works on energy-harvesting hardware such as WISP, Moo, etc. and its software design space: intermittent programming models such as Alpaca, Chain, etc. and task-based compilers like CleanCut. With the increasing developer tool chains like CleanCut and Sloth, the developer community of intermittent computing is getting more and more diverse.

We present design space of intermittence in the aspects of ambient power derivatives and software design space. Five programs are used as benchmarks to evaluate Sloth and we use Moo to run the corresponding generated programs. Moo is a radio wave reader energy harvesting device which is programmable and is being adopted by most intermittent computing researchers. Moo is a derivatives of WISP, another prevalent intermittent testing device. In evaluation, we use the five open-sourced programs to verify the validity and performance of Sloth. The results show that all of the five generated programs execute successfully on Moo and output the program result correctly. The complexity of the five programs are increasing from one to five. The experiment shows that if the more complex that the program is, the longer compilation time Sloth needs to process it. After evaluation, we discuss limitations and challenges of Sloth and its related works. Sloth is not perfect in the dimension of program compatibility because it only supports single file C program written in functional programming style.

Moreover, from the application level, engineers can optimize the performance of power-efficient software by categorize it with normal mode (Active Mode) or low power mode (Idle Mode / Sleep

Mode). The normal working mode here refers to the mode that can realize the normal system function. In this mode, the MCU is not always in the working state, but combines the active and Sleep MCU states according to the power consumption formula mentioned in the previous chapter. Mode. In short, a mode of operation that guarantees the application of basic functions while sleeping and sleeping. On the other hand, the low-power mode mentioned here refers to shutting down unnecessary functions as much as possible according to the user's application requirements, when the user specifies or the system automatically detects certain conditions, and only retains the tasks necessary for some applications. A mode that reduces power consumption. For example, turn off the LCD and LED backlight without user operation for a while. In summary, the normal working mode and the low power mode here do not directly correspond to the Active and Sleep modes provided by the MCU, but two power modes with different functional configurations defined according to user functional requirements. They may all involve the Active and Sleep states of the MCU. These two definitions are clarified, and many of the statements we have involved in subsequent discussions will not be confused.

Moreover, some microcontrollers can configure the drive current and slew rate of the IO port. One possible way to optimize the intermittent environment is to select the appropriate drive current and slew rate based on the device connected to the IO port, which reduces the drive current and slew rate to reduce system power consumption. Taking full advantage of the microcontroller's sleep mode, even if the system is idle for a short period of time, it can be put to sleep, which can reduce the system activity cycle. Some peripherals still work in some sleep modes, and the proper sleep mode can be used to reduce power consumption depending on the particular application. If the flash memory of the microcontroller has spare space, the program code can be optimized to improve the program execution speed. This way the task can be completed faster and the system will have more time to sleep. In this way, the power consumption of a certain extremely low-power device will significantly decrease and make the intermittent computing more efficient and reliable.

Besides, there are multiple advantages of the remote communication module: compact NB-IoT, NB-IoT/eMTC (Enhanced Machine-Type Communication) mobile communication module, ultra low power consumption, ultra high sensitivity, LCC (Leadless Ceramic) package for mass production, package design compatible with GSM (Global System for Mobile Communications) / GPRS (General Packet Radio Service) module for remote communication, easy product upgrade, embedded rich network service stack, meet customer's fast time-to-market requirements by providing reference designs, evaluation boards, and timely technical support. For example, a milk analyzer that interpret the test results of fat, protein and density, etc. of dairy cattle [18]. Also, the ultra-low power consumption of the RSL10 is ideally suited for a completely battery-free solution through energy harvesting. The energy collection scheme refers to the system itself converting energy such as mechanical energy or light energy into electrical energy, without wires or battery power, and is suitable for application scenarios that are difficult to maintain. The compiler can generate object code that runs in the same environment as the computer and operating system (platform) on which the compiler itself is located. This compiler is also called a "local" compiler. In addition, the compiler can also generate object code for running on other platforms. This compiler is also called a cross compiler. Cross compilers are very useful when building new hardware platforms. "Source to source compiler" refers to a compiler that uses a high-level language as input and a high-level language. For example: Automated parallelization compilers often use a high-level language as input, convert the code, and annotate it with parallel code comments or with language constructs.

We believe Sloth gives incremental value to the field of software intermittence. We also conclude some experience on developing an intermittent compiler or developer toolkits. When developing an automation tool, we should mainly run the Bucket Testing or A/B Testing, such as C language, there will be compiled classes, execution classes, performance classes, and so on. If you compile the class, you will be able to compile it, whether it will report the specified information, etc. This step is mainly used for testing at the syntax level. The running class mainly refers to whether the result of running out meets expectations, such as the final result, the return value, and so on. Performance class, mainly the time spent running this program, and even memory, etc., generally run the spec, this will generally run for a long time, we run spec are to open a screen, and then let it run automatically. All in all, the correctness of the compiler is the most vital thing to retain. In the future, we should make Sloth compatible with programming styles other than functional programming and include the functionalities of supporting interdependent classes or objects and other programming languages other than C. The task efficiency should be verified and the difference among compilation computer and actual intermittent computer should be included in the implementation of Sloth.

REFERENCES

[1] 01.org, “Powertop”, https://01.org/powertop/overview [Accessed: 22 July 2019].

[2] asterisk.org, “Asterisk Communications Framework”, [Online] https://www.asterisk.org/ [Accessed: 22 July 2019].

[3] beginnersbook.com, “C Program to check if a number is palindrome or not”, [Online] https://beginnersbook.com/2015/02/c-program-to-check-if-a-number-is-palindrome-or-not/ [Accessed: 22 July 2019].

[4] beginnersbook.com, “Program to convert octal to binary”, [Online] https://beginnersbook.com/2017/09/c-program-to-convert-octal-number-to-binary-number/ [Accessed: 22 July 2019].

[5] G. Cohn, S. Gupta, T. J. Lee, D. Morris, J. R. Smith, M. S. Reynolds, D. S. Tan, S. N. Patel, “An ultra-low-power human body motion sensor using static electric field sensing,” In Proceedings of Ubicomp (2012).

[6] A. Colin and B. Lucia. “Chain: Tasks and channels for reliable intermittent programs,” In OOPSLA, 2016.

[7] A. Colin, B. Lucia. “Termination checking and task decomposition for task-based intermittent programs,” In Proceedings of the 27th International Conference on Compiler Construction (CC 2018). ACM, New York, NY, USA, 116-127.

[8] J.A. Duffie, W.A. Beckman, “Solar energy thermal processes”, United States: N. p., 1974. Web.

[9] C. Gomez, J. Oller, J. Paradells, “Overview and Evaluation of Bluetooth Low Energy: An Emerging Low-Power Wireless Technology,” Sensors 2012, 12, 11734-11753.

[10] github.com, “jart/asterisk-voicechanger”, [Online] https://github.com/jart/asterisk-voicechanger [Accessed: 22 July 2019].

[11] github.com, “louis-xu-ustc/MotionDetection”, [Online] https://github.com/louis-xu-ustc/MotionDetection [Accessed: 22 July 2019].

[12] J. Hester, K. Storer, and J. Sorber. “Timely Execution on Intermittently Powered Batteryless Sensors,” In SenSys, 2017.

[13] R. Kanan, O. Elhassan, “A Combined Batteryless Radio and WiFi Indoor Positioning for Hospital Nursing,” Journal of Communications Software and Systems, Vol. 12 No. 1, 2016.

[14] S. Keijzers, “Energy Consumption Analysis of Practical Programming Languages,” Radboud University Nijmegen, Master Thesis, 2014.

[15] U. Liqat, Z. Bankovic, P. Lopez-Garcia, M.V. Hermenegildo, “Inferring Energy Bounds via Static Program Analysis and Evolutionary Modeling of Basic Blocks,” eprint arXiv:1601.02800. 2016.

[16] B. Lucia and B. Ransford. “A simpler, safer programming and execution model for intermittent systems,” In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2015, pages 575–585, New York, NY, USA, 2015. ACM.

[17] K. Maeng, A. Colin, and B. Lucia. “Alpaca: Intermittent execution without checkpoints,” In OOPSLA. ACM, 2017.

[18] quectel.com, “EKOMILK STANDART,” [Online] https://www.quectel.com/ [Accessed: 22 July 2019].

[19] D. Roggen, A. P. Yazdan, F. J. O. Morales, R. J. Prance, H. Prance, “Electric field phase sensing for wearable orientation and localisation applications,” In Proceedings of the 2016 ACM International Symposium on Wearable Computers, Pages 52-53.

[20] A. P. Sample, D. J. Yeager, P. S. Powledge, A. V. Mamishev, and J. R. Smith. “Design of an RFID-Based Battery-Free Programmable Sensing Platform,” IEEE Transactions on Instrumentation and Measurement, 57(11):2608–2615, Nov. 2008.

[21] M. Santos, J. Saraiva, Z. Porkoláb, D. Krupp, “Energy Consumption Measurement of C/C++ Programs Using Clang Tooling,” SQAMIA. 2017.

[22] J. Singh. “Performance Analysis of RF Energy Harvesting Circuit with Varying Matching Network Elements and Diode Parameters,” 10.13140/RG.2.1.4838.4488. 2015.

[23] studytonight.com, “Program to perform addition and subtraction of Matrices”, [Online] https://www.studytonight.com/c/programs/array/addition-and-subtraction-of-matrices [Accessed: 22 July 2019].

[24] V. Talla, B. Kellogg, S. Gollakota, J. Smith, “Battery Free Cellphone,” IMWUT June 2017.

[25] V. Talla, S. Pellerano, H. Xu, A. Ravi, Y. Palaskas, “Wi-Fi RF Energy Harvesting for Battery-Free Wearable Radio Platforms,” IEEE RFID 2015.

[26] Y. K. Tan, K. Y. Hoe and S. K. Panda, "Energy Harvesting using Piezoelectric Igniter for Self-Powered Radio Frequency (RF) Wireless Sensors," 2006 IEEE International Conference on Industrial Technology, Mumbai, 2006, pp. 1711-1716. doi: 10.1109/ICIT.2006.372517.

[27] J. V. D. Woude and M. Hicks. “Intermittent computation without hardware support or programmer intervention,” In OSDI, 2016.

[28] R. Yazdani, A. Segura, J. Arnau and A. Gonzalez, "An ultra low-power hardware accelerator for automatic speech recognition," 2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), Taipei, 2016, pp. 1-12. doi: 10.1109/MICRO.2016.7783750.

[29] H. Zhang, J. Gummeson, B. Ransford, and K. Fu. “Moo: A batteryless computational rfid and sensing platform,” Department of Computer Science, University of Massachusetts Amherst., Tech. Rep, 2011.