HALMSTAD UNIVERSITY

Master's Programme in Embedded and Intelligent Systems, 120 credits

MASTER THESIS

A Simulation Based Approach to Estimate Energy Consumption for Embedded Processors

Liu Ke

Parallel Computing, 15 credits

Halmstad 2015-09-07

A Simulation Based Approach to Estimate Energy Consumption for Embedded Processors

Liu Ke

Supervisor:
Zain Ul-Abdin
Co-supervisor:
Essayas Gebrewahid

This thesis is presented as required for the conferral of the degree:

Master

Halmstad University
School of Information Technology

September 2015

Abstract

Embedded systems have entered a new era in which system designers have to consider more and more strict energy consumption constraints. This thesis reviews previous studies of the processor energy consumption estimation. Particularly, we focus on instruction-level energy consumption for embedded processors and explore the energy consumption model for manycore architecture in real traffic pattern. The purpose of this thesis project is to estimate energy consumption and construct an energy model using an instruction-set simulator for embedded processors. Open Virtual Platforms (OVP) and Epiphany Single Core Simulator (ESCS) are used to obtain an instruction sequence for a given software. Then, the functionality of energy consumption estimation is integrated into OVP.

Our energy consumption estimation approach categorizes instructions in four groups and uses base energy cost of each category to calculate the total energy consumption for an application that runs on Epiphany core and ARM Cortex M0. The energy consumption estimation for Epiphany core has been successfully tested by BEEBS benchmarks. Details of the derivation process, results and analysis of energy consumption estimation methods are provided.

Acknowledgments

I would like to thank my supervisor, Dr. Zain, for his support and thank my co-supervisor Essayas who sets a good example to me in my future life. I would also thank to Dr. Dariusz who did not abandon me and is still willing to be my supervisor at Wroclaw University of Technology. I have several other people I would like to thank, as well. They are my parents, my cousin, Professor Mohammad, Dr. Jan.Kwiatkowski from Wroclaw University of Technology, George B. P. Bezerra from MIT, and Duncan Graham from Imperas Software.

Contents

Abstract **ii**

1 Introduction **1**
1.1 Goal . . . 2
1.2 Contributions . . . 2
1.3 Organization . . . 2

2 Background Information **4**
2.1 Modern Computer Architecture . . . 4
2.2 Supporting Technologies . . . 10

3 Factors Causing Power Consumption in Processors **12**
3.1 Factors Causing Energy Consumption in Single-Core Processor . . . 12
3.1.1 Circuit level . . . 12
3.1.2 Functional level . . . 13
3.2 Factors Causing Energy Consumption in Network on Chip . . . 16

4 Abstraction Levels of Energy Consumption Estimation **18**
4.1 Source-Code Level . . . 18
4.2 Functional-Level . . . 19
4.3 Instruction-Level . . . 19
4.4 RT-Level (Register-Transfer abstraction level) . . . 19
4.5 Gate-Level . . . 20
4.6 Transistor-Level . . . 20

5 Energy Consumption Estimation **21**
5.1 Literature Review of Instruction-Level Energy Estimation . . . 21
5.1.1 The method of obtaining energy consumption data . . . 23
5.1.2 Energy consumption model . . . 24
5.1.3 Instructions acquisition approaches . . . 30
5.1.4 Overall energy consumption estimation flow . . . 30
5.2 Simulation Methodology of OVP . . . 31

5.2.1 Loop bound analysis 32
5.2.2 Path analysis 34
5.2.3 Control-flow analysis 35
5.3 Instruction-Level Energy Consumption Model Derivation Process . . 35
5.4 Overall Energy Estimation Flow 40
5.5 Energy Consumption Estimation Using Base Cost 43
5.6 Energy Estimation on Manycore Architecture 45

6 Case Study **48**
6.1 The Deriving Process of Energy Base Cost 48
6.2 Energy Consumption of Benchmarks and Analysis 51
6.2.1 The relationship between number of instructions and energy consumption 51
6.2.2 The relationship between cycles and energy consumption . . . 54
6.3 Validation Based on Datasheet of Epiphnay 57
6.4 Conclusion 59

7 Conclusion and Future Work **60**
7.1 Conclusion 60
7.2 Future Work 61

Bibliography **64**

A Codes For This Project **70**
A.1 Energy Model in OVP 70
A.2 C# Code for Calculating Energy Consumption of Benchmarks in Epiphany 77

B Tables of This Thesis **82**
B.1 Benchmarks energy consumption estimation and their instruction category details 82

Chapter 1

Introduction

Embedded systems are used in all aspects of human life and industries. Mobile phone, tablets, cars, robots, and computerized numerical controls are typical examples that integrate with an embedded processor. Besides, some of them (e.g, mobile phone) are portable battery powered devices that have limited power resource. Hence, most of embedded systems need to meet energy constraint. Both performance and energy consumption is one of the most important metric for embedded system design. Embedded system designer usually have to find the trade-off between performance and energy consumption. There are numerous application scenarios where system designer should take into account performance and energy consumption. Therefore, validation and estimation of performance and energy consumption are essential for embedded system design. Especially, an effective design exploration during the early stage for embedded system design saves both time and development expenses. This motivation drives us to study energy consumption estimation in the early design stage using an instruction set simulator such as OVP (Open Virtual Platforms) and ESCS (Epiphany Single Core Simulator).

The emergence of full-system simulators provide the possibility to execute unchanged production binary code of the target hardware on a simulator without target hardware in the early design stage. Thus, full-system simulator is a suitable choice for energy consumption estimation. In general, a full-system simulator is utilized to obtain functional requirement metrics of the system. Yet, non-functional requirement metrics such as energy consumption estimation are often neglected. Thus, empowering the state-of-the-art simulator to estimate energy consumption is significant to embedded system design.

This thesis focuses on modeling and estimation of energy consumption in embedded processors. We present some energy models to estimate energy consumption and integrate the model into a widely used simulator OVP. Besides, we also use another dedicated simulator for Epiphany processor to estimate energy consumption and validate the proposed model. Finally, we give our insight to extend the energy

consumption estimation solution for manycore architecture.

1.1 Goal

The goal of this thesis is to provide a fast accurate simulation based energy consumption solution for a given application on embedded processors in the early design stage.

1.2 Contributions

Our main contributions can be summed up in these points.

- We introduce an overview and a classification of previous studies about instruction-level energy consumption estimation.
- We propose a series of improved energy models based on previous studies and design the flow of the energy estimation of OVP at instruction-level.
- We demonstrate the derivation of energy base cost can be used to estimate total energy consumption within acceptable accuracy for certain processors.
- We propose the energy consumption estimation model for the Epiphany manycore architecture.

1.3 Organization

The remaining of this thesis contains seven chapters.

Chapter 2. This chapter contains a review of the background material required for the thesis. We give a brief introduction to computer architecture, basic manycore architecture, and some third part tools which are used in our project.

Chapter 3. This chapter analyzes factors causing energy dissipation in an embedded processor from different abstraction levels.

Chapter 4. In this chapter, we review previous studies related to energy consumption estimation and classify them into different abstraction levels. By analysing the merits and drawbacks of these energy estimation approaches, the instruction-level energy estimation is selected for our energy consumption estimation model.

Chapter 5. This chapter gives a literature review of instruction-level energy estimation. In addition, we deeply analyze some typical instructional-level energy studies with their characteristics and also examine them from different perspectives. We elaborate the energy model derivation process and propose the overall estimation flow. Also we demonstrate the accuracy if only the base energy cost is used to estimate energy consumption for simple pipelined, in-order execution processors. Finally, we present the derivation process of the energy consumption estimation model in Epiphany manycore architecture.

Chapter 6. Here, we present the energy consumption results using the base cost for Epiphany core and ARM Cortex M0. This chapter also validates the accuracy of base energy cost using the BEEBS benchmarks [1].

Chapter 7. This chapter concludes the thesis, and proposes some suggestions for future work.

Chapter 2

Background Information

This chapter covers the background information and supporting technologies required in the succeeding chapters. Section 2.1 reviews instruction set architecture, pipeline, and manycore architecture. Section 2.2 introduces two tools which are used to obtain instructions for a given application and a programing language which is used to analyse instructions.

2.1 Modern Computer Architecture

Computer architecture is the fundamental knowledge to understand simulation based energy consumption estimation. Modern computer comprises advanced ISA, multi-stage pipeline, out of execution, and multi-core or manycore. Hence, we briefly review some of the key concepts required for this thesis.

Instruction Set Architecture (ISA)

Instruction Set Architecture is the lowest level of computer architecture that is visible for a compiler writer or a programmer. It can be viewed as a bridge for high-level programming languages such as C, C++ to the set of opcodes, native commands as well as processor registers, addressing modes and data formats implemented by a particular processor. One typical instruction may contain several fields, such as opcodes, result operand, source operands, immediate value, or even condition field, control field, etc. One instruction also implicitly denotes the addressing mode, which is the way of accessing the address of the operands. Herein, we use ARM instruction set as an example to interpret the concept of ISA, Table 2.1 shows several instructions and the explanation of these instructions. It is worth noting that the ISA of ARM and most of the state-of-the-art processors (MIPS, Epiphany, etc.) are the *load/str* architecture, which means only memory operation instructions (*ldr, str etc.*) can access the memory. Instructions such as "*add*" can only use operands in

Instruction	opcode	result operand	source operand	meaning	addressing mode
add r3, r2, r3	add	r3	r2,r3	R[r3]=R[r2]+R[r3]	Registe
add r3, r3,#1	add	r3	r3,#1(I)	R[r3]=R[r3]+1	immediate
ldr r0, [r1]	ldr	r0	r1	R[r0]=M[r1]	register indirect
ldr r0, [r1, #4]	ldr	r0	r1,#4(I)	R[r0]=M[r1+4]	register indirect with offset
ldrsb r0, [r5, r1, lsl #1]	ldrsb,lsl	r0	r5,r1,#1	R[r0]=M[r5+2*r1]	double Reg indirect with scaling

Notes:
#1: the immediate value 1, R[]: Register value, M[]: Memory value
R[r0]=M[r5+2*r1]: Read value from memory and store in r0, the address of memory is calculated as value in r5 added by 2 times value in r1.
lsl: one ARM instruction, which means logical shift left. In other words, lsl #1 means times 2

Table 2.1: ARM instructions

the registers. For example, in Table 2.1, the instructions typically belong to *load-/str* architecture. For instance, the instructions such as "*add r0, r2, [r1]*" do not exist in *load/str* architecture. Furthermore, the ISA of modern microprocessors can be classified as Reduced Instruction Set Computer (RISC) and Complex Instruction Set Computer (CISC). Just as its name implies, RISC processors have simpler and fewer instructions than CISC processors and utilize fewer clock cycles than the CISC processors. Our target are embedded processors, therefore this thesis mainly discuses on RISC processors.

Simple pipeline and superscalar pipeline

For the state-of-the-art processors, the pipeline is an essential technique used in design for modern processor to enhance performance by means of overlapping executing instructions. In order to properly interpret the necessity and principle of the pipeline, let's assume a processor without pipeline. So the instruction sequences must be executed one by one as shown in Figure 2.1.

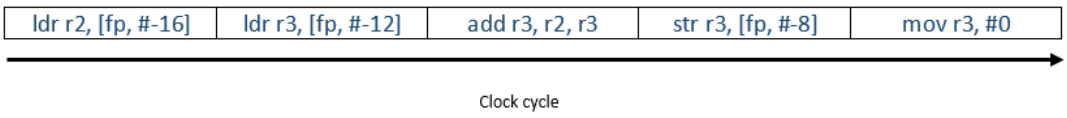


Figure 2.1: Five instructions sequence executed in a non-pipelined processor

As can be seen in Figure 2.1, five instructions (ARMv6 instructions) are executed in a non-pipelined processor sequentially. Starting from the second instruction, each instruction has to wait until the previous instruction is to be finished. Therefore, utilization of the processor component would be ineffective. For example, when one instruction is being processed in the arithmetic logic unit (ALU), the load instruction unit of the processor is being laid idle.

Therefore, the instruction pipeline is implemented to overcome above problem. In a pipelined processor, each instruction is divided into some consecutive pipeline stages so that instruction sequence can be executed in parallel. A classic RISC pipeline is shown in Figure 2.2, the instruction sequence is denoted from top to the

bottom and the horizontal axis represents clock cycle increment. The overlapping of

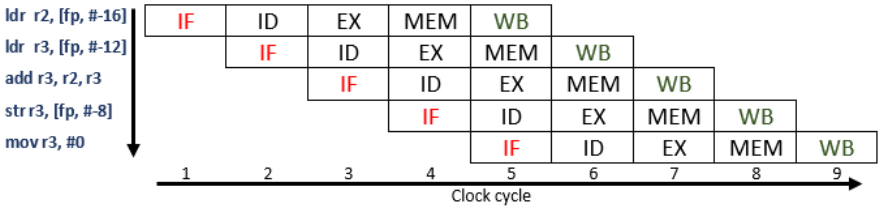


Figure 2.2: Five instructions sequence executed in a five-stage pipelined processor

instructions is distinctly illustrated in Figure 2.2. For instance, there are 5 instructions activated in clock cycle 5, the first instruction "*ldr*" is in its WB (Register write back) stage while the second instruction "*ldr*" is in its MEM (Memory access) stage. Simultaneously, the third instruction "*add*" is in its instruction EX (Execute) stage and the fourth instruction "*str*" is in its ID (Instruction decode and register fetch) stage. Again, the last instruction "*mov*" is being fetched, namely, in IF (Instruction Fetch) stage. Note that the execution time of single instruction executed in a pipelined processor is probably to be prolonged compared with non-pipelined processor. Yet, the total execution time of an instruction sequence is shortened by increasing the throughput of instructions. Therefore, the more instructions are executed in the pipeline, the more efficient utilization of the pipeline could be gained. However, pipelined processors like shown in Figure 2.2 are hard to achieve the ideal throughput. The performance lost are, fairly often, due to unnecessary stalls which are caused by structural hazards, data hazards or control hazard. Structural hazards occurs when one instruction compete with another instruction for the same processor's hardware resource; data hazard is happened when the source operand of one instruction is the result operands of an earlier instruction. Hence, the later instruction need to wait until the earlier instruction is completed; control hazard (aka branch hazards) refers to whether or not an instruction can be executed is determined by an earlier branching instruction.

Although the classic five-stage pipeline is used to illustrate the concept of a pipelined processor. In practice, the number of pipeline stages is various for commercial microprocessors. For instance, ARM7 and earlier implementations have a three-stage pipeline i.e., fetch, decode and execute. Higher-performance processors, such as ARM9, ARM10, and ARM11, have five-stage, six-stage, and eight-stage pipeline respectively [2]. Besides, Adapteva's[a] Epiphany core employs a variable length eight-stage pipeline, where the terminology variable length pipeline denotes several pipeline stages are unused by certain classes of instructions. For instance, *load/store* and floating pointing instructions do not go through last three pipeline

[a]Adapteva is a fabless semiconductor company, which designed Epiphany.

stages [3].

So far we have already discussed the simple pipeline and its limitation. The above simple pipeline can only issue one instruction per clock cycle. By contrast, a superscalar pipeline can issue more than one instruction at a given clock cycle so that multiple instructions can be dispatched to different functional units. Figure 2.3 illustrates a simple dual-issue superscalar pipeline, which can issue and execute double instructions per clock cycle.

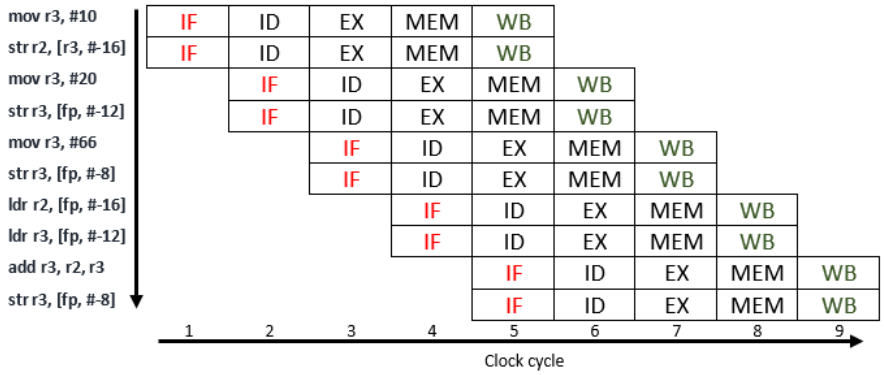


Figure 2.3: Five instructions sequence executed in a five-stage superscalar processor

Though the instructions issuing rate is increased, superscalar pipeline still suffers from the above-mentioned hazards. As shown in Figure 2.3, the first instruction "*mov r3,#10*" and the second instruction "*str r2, [r3,#-16]*" cannot be executed in parallel because of a data dependence between them.

Out-of-Order execution

The execution order of the instruction sequence for a pipelined processor has two ways. One is in-order execution, and another one is out-of-order execution (aka dynamic execution). Basically, in-order execution processors fetch, execute and complete instructions in accordance with compiler generated-order. On the other hand, out-of-order execution need to rearrange the execution sequence. However, the instruction fetching order is still the same as compiler generated-order and the instructions completion order could be either in-order or out-of-order.

The out-of-order execution intends to mitigate hazards and undesirable stalls without changing the outcome or functionality of the instruction sequence. As mentioned previously, hazards and undesirable stalls are a principal limitation of in-order pipelined processors. Therefore, most of modern high-performance processors are implemented with out-of-order execution. However, this does not mean that in-order execution is a technique which is no place to use. Many state-of-the-art embedded processors still employ in-order execution. For example, Adapteva claims

that out-of-order execution is rejected for their latest Epiphany processor because this technique cannot significantly boost performance in streaming signal processing applications and energy efficiency concern [3].

The performance enhancement of out-of-order execution comes from two improvements compared with in-order execution. First, a class of stalls (e.g., "*ldr*" instruction takes many clock cycles to wait memory write back so subsequent instructions have to be stalled) are avoided through previous instructions can be executed after subsequent instructions if there is no true dependency between them. It is noteworthy that instructions are decoded (also fetched as mentioned before) in-order so that WAR (Write After Read) dependence is eliminated. Besides, the WAW (Write After Write) dependence is resolved via renaming their register operands. Second, stalls that are incurred by branch hazards are also reduced since instructions after branching instructions can be speculatively executed before the branch is determined. Of course, if the speculation is incorrect, speculative instructions have to be abandoned and roll back to the starting point of the right branch path. In addition, the out-of-order execution processor involves many additional hardware implementations (i.e., reservation station registers, common data bus, and re-order buffer). Incidentally, the instruction sequence order can also be rescheduled in code compilation stage contrast to the above-mentioned hardware based out-of-order execution. This kind of processors is called as Very Long Instruction Word Processor (VLIW). Since our discussion will not involve too many details of out-of-order execution hardware implementations and VLIW, more details please refer to [4].

Manycore

The term manycore most commonly refers to a single chip which contains more than 8 independent cores. In contrast, multi-core means a single chip consists of 2-8 cores. Yet, there is no rigorous definition in terms of the number of cores on a single chip for manycore and multi-core. Herein, we just provide a common view as a reference.

Whatever manycore and multi-core are, cores on a single chip could be either identical or not. If cores are identical, these kinds of processors are called homogeneous, otherwise they are heterogeneous. For example, nowadays one of the most widely used homogeneous multi-core processors are Intel I-series CPUs. Another typical homogeneous processor is Epiphany core which contains 16 or 64 identical E16G301 cores or E64G401 cores.

Multi-core and manycore processors may implement common network topologies to interconnect cores such as bus, network on chip, and crossbar for data communication. Compared with network on chip, the on-chip bus works efficiently for a small number of cores on a chip. However, as the number of cores on a single chip

increases, the limitations of on-chip bus have been revealed. Several drawbacks of on-chip bus are summarized as follows:

- The performance of data transmission decreases, as the number of processors increases. For example, collisions, and starvation are likely to occur.
- Due to the centralized topology, once the bus encountered a problem, the whole system breaks down.
- As the number of functional units and the size of the bus increased, the energy consumption for switching a bit grows significantly.

However, network on chip brings a much more communication resources, better scalable and energy efficiency. Therefore, on-chip bus has a tendency to be replaced by a network on chip as the number of cores on a chip increasing. There are many commercial products that have been launched such as Adapteva's Epiphany core, Ambric's Am2045, and Intel Xeon Phi etc.

As mentioned earlier, the functionality of a network on chip of manycore processor is to send and receive message among Intellectual Property (IP) cores in a System on a Chip (SoC). A single node of the network on chip usually consists of several factors, namely, routing, arbitration and switching. Briefly speaking, routing defines what kind of path a packet could pass through. Arbitration defines when to allocate the path for a packet. Switching defines how connection put through. Each single node and channel in the network on chip are statically arranged by network topology. Figure 2.4 depicts the structure of basic 2D network on chip. Note that, the Network Interface (NI) provides the interface between the IP and the network, which takes the charge of packetization and de-packetization of messages, clock adaptation, flow control, and implementing the network protocols [5]. Incidentally, the network on chip of Epiphany is built on 2D network.

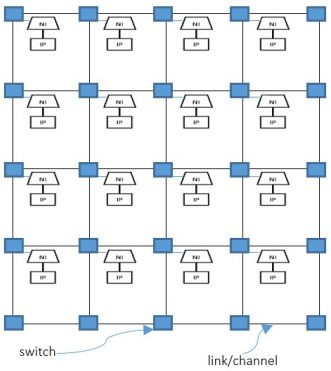


Figure 2.4: Basic Network on chip structure (Figure reproduced from [6])

Besides, network on chip community has proposed many network topologies such as fat tree, torus, folded-torus, octagon, and butterfly fat tree, etc.

2.2 Supporting Technologies

The following sections introduce any third party resources that we have used during our project.

Open Virtual Platforms

Open Virtual Platform (OVP) is a typical full system simulator. It aims to run application code on simulator without any modification. In other words, OVP provides the virtual computer which does not depend on the nature of the host computer. Programmer needs to create a platform, which constructs processor cores, caches, memory and bus as a virtual platform for application. Figure 2.5 depicts C interfaces of OVP.

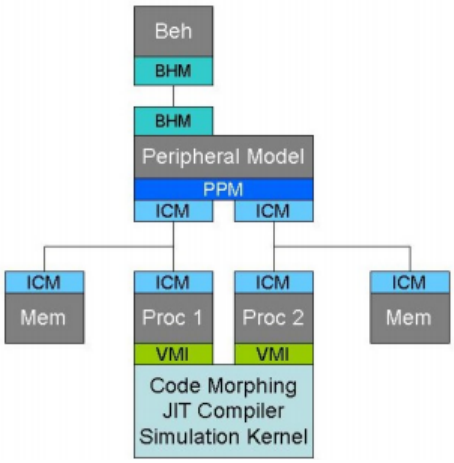


Figure 2.5: OVP interface given by [7]

Among these interfaces, BHM (Behavioral Modeling) can be used to create a thread, event object, delays, diagnostic text output etc. ICM (CpuManager Interface) is responsible for creating a platform for memory, processor, peripherals and other hardware blocks. The ICM is a C interface that when compiled and linked with those of each models and some object files produce an executable model [8]. VMI (Virtual Machine Interface) is to create processor models for use with CpuManager and OVPsim[b]. Imperas[c] states that processor models created by VMI can achieve hundreds of millions of simulated instructions per second [9]. Therefore, VMI can be viewed as the key component of OVP. The essence of OVP simulation is coupled with a just-in-time compiler to map ISA of the simulated processor into ISA of the host processor [7]. This procedure is called as code morphing approach by Imperas.

[b]CpuManager and OVPsim are paid and free versions of OVP respectively.
[c]Imperas software is the owner of OVP.

The reason why we chose OVP is that we already have user experience before this project. Moreover, based on our extensive testing and verification of OVP, it is indeed an instruction accurate simulator as Imperas claims. More details about the conclusion of instruction accurate for OVP will be discussed in Chapter 5.

Epiphany Single Core Simulator (ESCS)

Epiphany Single Core Simulator (ESCS) is a part of in Epiphany Software Development Kit (SDK). Adapteva claims that the simulator is a fast functional simulator with instruction accurate trace capability while the register map of a single Epiphany core can also be obtained [10]. This simulator cannot model the network-on-chip of Epiphany (eMesh), pipeline, timing behavior and DMA (Direct Memory Access). Incidentally, a third party [11] also provides the Epiphany simulator that can model eMesh. The reason why we chose Epiphany simulator is because it perfectly meets our instruction accurate requirement. Besides, OVP does not support the Epiphany model at present. Note that Epiphany SDK can install on all mainstream operating systems (Windows, Mac OS, Linux, et.), the simulator runs in a host Linux environment. Thus, non-Linux based OS requires Unix-like environment such as Cygwin.

The following example demonstrates how to use Epiphany compiler to obtain *elf* file from an application. After obtaining the *elf* file of the application, the command

```
$ e-gcc adpcm.c -o adpcm.elf
```

e-run can be used to simulate the application on the host computer as the following example shows. Note that option '-t' means that to trace and output instructions.

```
$ e-run -t adpcm.elf
```

C#

C# is our second chosen programming language to classify instructions. We will be using the Visual Studio IDE (Integrated Development Environment) to work in C# to utilise its advantages. This C# work consists of several functions which performs string analysis for text of instructions and creates excel file for linear regression.

Chapter 3

Factors Causing Power Consumption in Processors

In order to estimate energy consumption for a given application on an embedded processor, it is necessary to know how energy is consumed when a program is executed on an embedded processor. Energy consumption in a processor depends on a variety of factors. Also, it is difficult to analyze the dependence of each factor on energy consumption. In this chapter, we present the different perspectives to analyze the energy consumption to an embedded processor.

3.1 Factors Causing Energy Consumption in Single-Core Processor

Factors causing energy consumption could be interpreted from different abstraction levels. From the view of the digital circuit, we briefly explain the energy consumption in a processor. Then, we analyze main energy consumers on a processor as well as off-chip energy consumers on an embedded system. Finally, we summarize some major factors of processor design and runtime environment that could affect energy consumed by a processor.

3.1.1 Circuit level

The total energy or power consumption of a digital CMOS (Complementary Metal Oxide Semiconductor) circuit consists of two components, namely, static energy (power) dissipation and dynamic energy (power) dissipation. It can be expressed as:

$$E = E_{static} + E_{dynamic} \tag{3.1}$$

where E_{static} is the energy dissipation that occurs when CMOS is in standby mode, generally regarded as waste of energy. Leakage current causes static dissipation. The leakage current has five components, namely, reverse biased pn junction current, sub-threshold leakage, gate induced drain leakage, punch through, and gate tunneling [12]. However, for properly designed CMOS processor, this static dissipation is often insignificant but it becomes significant when the transistors become smaller and faster [13].

The dynamic part of energy dissipation accounts for most of the energy dissipation on CMOS circuit, It is composed of two terms:

$$E_{dynamic} = E_{swiching} + E_{short-circuit} \tag{3.2}$$

where $E_{swiching}$ is energy consumed when the load capacitance are charged and discharged associated with each node in CMOS circuit [13]. It is given by:

$$E_{swiching} = \frac{1}{2} * C * v_{dd}^2 \tag{3.3}$$

where C is the effective load capacitance, and v_{dd} is the supply voltage of the circuit. The second term of Equation 3.2 is incurred by short circuit current on which both PMOS and NMOS networks are conducted [13]. Studies [14], [15] show the switching activity dominates the energy consumption by making up to 90% of the total consumption. Thus, the key to energy consumption estimation is to find the switch activity energy dissipation.

3.1.2 Functional level

First, we list main energy consumers that are functional units in an embedded processor. Then we present several non-functional factors which affect the energy consumption on these components of a processor.

Datapath

A datapath is a group of functional units, such as arithmetic logic units or multipliers that perform data processing operations, registers [16]. Any operation in datapath can reflect energy consumption because it causes switching activity in that particular component contributing to the dynamic energy consumption [17].

Cache

Cache is the fastest and the smallest on-chip memory to a processor. One cache line usually contains three components (index, tag, and data) and it is used to store the corresponding contents of memory as shown in Figure 3.1.

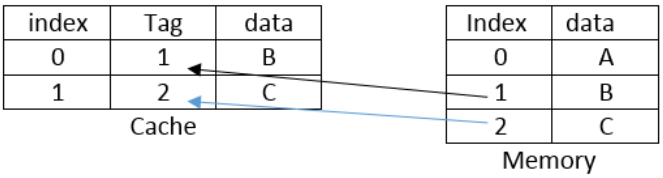


Figure 3.1: Memory cache operation

When we analyze energy consumption in cache accessing, more details of cache should be taken into consideration. As cache architecture (Figure 3.2) shows, when

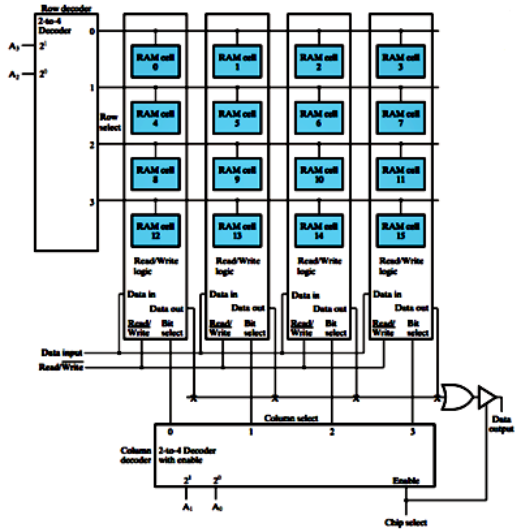


Figure 3.2: Cache architecture. Figure reproduced from [18]

reading\writing request comes, row and column decoder into both row and column select signals. After that, address is decoded into a set of wordlines. Wordlines select row to be read\written. Apparently, all of these procedures incur energy consumption due to the dynamic energy dissipation. In study [19], cache is regarded as one of the greatest energy consumer on processor, which accounts for 43% of total energy consumption.

Other off-chip components

There are two main energy consumers for an embedded system which are outside of the processor. If the target of energy consumption estimation is the entire embedded system, these components cannot be neglected.

Bus: The bus is a communication system that is responsible for transferring data between processor and cache, cache and memory or memory and peripheral. Note that bus could be in the processor. For example, a bus is used to connect manycore to shared memory in manycore architecture. Whenever the data are

transferred on the bus, the energy consumed by bus is caused by switching activity.

Main Memory: Although many embedded systems have a limited on-chip memory, namely, caches. However, when a complicate or large task comes, it cannot avoid using external memory. Memory access is an energy hungry operation since external memory accessing which incurs longer duration of latency and execution time has lower speed than cache. This long duration causes significantly energy consumption increased. Moreover, due to distinctive principle of implementing DRAM and SRAM. The energy consumed for writing and reading should be treated differently in DRAM.

Other Factors

Factors which affect energy consumption on energy consumer for the processor are variety and complex. Energy consumption can be influenced by design details of the processor. Some important factors are summarized as following:

Addressing model: Addressing model influences on the energy consumption because the variation of addressing model has different duration of calculating memory address and using different number of registers.

Register file size: The register file is an array of registers on a processor, a number of register are allocated while executing an application. Typically, as the size of the register file increases, the area of the chip is increased as well. Hence, increased chip area will cause the energy consumption of the processor.

Clock frequency: The energy consumption increases with increasing clock frequency. The dynamic power consumed by a processor is approximately proportional to the CPU frequency, and to the square of the processor voltage (Equation 3.4) [20].

$$P \simeq C * V^2 * f \tag{3.4}$$

Thus, some techniques such as dynamic frequency scaling could be employed to adjust the frequency of the processor "on the fly" so that energy consumption and heat dissipation could be decreased.

Branch Misprediction: Modern processors widely implement the branch predictor to reduce the branch penalty associated with conditional instruction execution. When a branch misprediction occurs, the branch instruction has to have a penalty of a few clock cycles. Branch misprediction is one of the reasons incurs the pipeline hazards that introduce the pipeline stalls, therefore, energy is wasted when pipeline stall happens.

Out-of-Order Execution: Out-of-order execution is another widely used technique to improve the performance of processors. The core idea of out-of-order execution is to reorder instruction sequence without changing the result or functionality of this instruction sequence to mitigate impact of data hazards so that the total execution cycles and data hazards penalty could be reduced. It is unknown whether predictor and out-of-order execution would increase the energy consumption or not. On the one hand, reduced pipeline stalls decrease the energy consumption, yet additional components will incur more energy consumption.

Runtime Environment: Apart from the hardware design details, runtime environment affects the energy consumption on the target processor as well. For instance, operating system was found that on an average; around 32% of the total energy was drawn by OS alone for a particular workload [21]. Besides, the program execution time significantly impacts on energy consumption in processors. As more instructions are executed, the processor is probably to spend more clock cycles. Hence, the energy consumed on the target processor is partially determined by the program execution time. Furthermore, input data values determine the branch target or loop boundary for an application task. Thus, the total number of instructions varies. In other words, input values influence the execution time, and thereby impact the energy consumption.

3.2 Factors Causing Energy Consumption in Network on Chip

Since our future work is to estimate the energy consumption for manycore processor, it is important to find out how energy is consumed on network on chip. Based on architecture of network on chip, we can infer that the total energy consumption consists of the energy consumed by each router and wires between routers. The total energy consumption is given by:

$$E_{networksonchip} = E_{router} + E_{wires} \tag{3.5}$$

Because the router in a network on chip is constituted by a CMOS circuit, the energy consumed is also composed of switching energy, leakage energy and internal energy, where the leakage energy is static energy consumption, the other two are dynamic energy consumption. Note that the term internal energy consists of aforementioned shortcircuit energy consumption and the switching energy consumption of internal

nodes. Therefore the total energy consumed in a router is given by Equation 3.6:

$$E_{router} = E_{switching} + E_{leakage} + E_{internal} \tag{3.6}$$

The energy model of a router for network on chip, as shown in Figure 3.3 , consists of the following parts: crossbar energy, clocking energy, FIFO buffers energy, arbiter energy, links energy, and MSINT. All of them contribute dynamic energy dissipation and leakage energy dissipation. Note that the proportion of each energy consumer which is shown in Figure 3.3 may be varied due to different types of routers and running environments such as supply voltage or temperature.

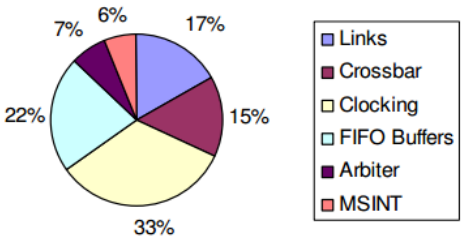


Figure 3.3: Router power breakdown at 4GHz, 1.2V, and 110 °C, given by [22]

.

Apparently, energy consumption model of a router in network on chip can be expressed as:

$$E_{router} = E_{Links} + E_{Crossbar} + E_{Clocking} + E_{FIFOBuffers} + E_{Arbiter} + E_{MSINT} \tag{3.7}$$

This energy model can be viewed as a functional-level energy model as we defined an energy consumption model at functional-level for a single core.

Chapter 4

Abstraction Levels of Energy Consumption Estimation

In general different abstraction level can be used to describe a processor and its behaviors from different perspective. Similarly, energy estimation could be performed at different abstraction levels. In this chapter, the brief introduction of energy estimation at different abstraction level is provided, highlighting their pros and cons.

4.1 Source-Code Level

Source-Code level is the highest abstraction level in which we could analyze energy consumption. Theoretically, source code level may be the easiest way to analyze energy consumption since it is easy to get the source codes for a programmer and the internal details of processor are hidden from high level languages. Furthermore, information about memory access, arithmetic and logical operations can be captured from any type of high-level programming language (e.g., Java, C++, and C etc.). However, there are some obstacles which impede to analyze energy consumption from the application source code level. Initially, there are many high-level programming languages and each high-level programming language often has many compilers. For example, C has more than 50 kinds of the compilers and compilers are often designed with optimization technique (e.g., inlining, loop unrolling as well as instruction scheduling) to improve performance on the specific target processor. Hence, we believe that source code analysis is difficult to establish relationship between the source code and the hardware operation. Finally, source code level energy consumption estimation cannot avoid analysing the loop boundary and execution path, which may pose extra challenges. For instance, study [23] analyzes path information from Java source code before estimating energy consumption. In recent times, research on source code level energy analysis has just started, only a few

proposals have been made at this level, thus, yet lacks a rigorous approach.

4.2 Functional-Level

We have already analysed the energy consumption of functional units on processor in Section 3.1. Therefore, it is natural to think if the energy dissipation of these functional units can be obtained, the total energy consumption of the processor is the sum of the energy dissipation of these functional units. The energy model of this level is also constructed by means of either simulation or direct measurements. The energy parameters for energy model of this level are the access rate of the on-chip memory, the clock frequency or the degree parallelism etc. Obviously, these parameters are dynamic values that can be determined after the execution of a given task. Theoretically, this level of energy consumption estimation can be applied to any type of processor. Besides the complexity of energy model of this level is moderate.

Moreover, Blume et al. [24] present a hybrid approach that combines functional-level and instruction-level power model to estimate energy consumption.

4.3 Instruction-Level

ISA is one of the ways to describe a microprocessor. Intuitively, ISA could be used to estimate energy consumption of processors since instructions relates to the functional units of the processor. Many studies present energy estimation approach at instruction level. This type of application energy consumption estimation usually relies on either instruction-level simulator or assembly code analysis to obtain instruction sequences of application. Instruction-level energy estimation has two major types. Early studies mainly focus on measuring energy cost per instruction without consideration any design details of processors; recent studies take into account pipeline details so that energy cost for each instruction could be measured and modeled at each pipeline stage. For more details of instruction-level energy estimation, please refer to Chapter 5.

4.4 RT-Level (Register-Transfer abstraction level)

RT-level abstraction level refers to digital signal flow between hardware registers, memory, bus, combinational logic devices and logic unit etc. VHDL (VHSIC Hardware Description Language), Verilog, and SystemC [14] are, fairly often, used to describe RT abstraction level. RT-level energy estimation is used to quickly predict the total switching activity in logic design compared to the simulation speed of

gate-level estimation. In this technique, switching activity profiles, circuit modules or control signals are integrated to construct macro-module which can be parametrized in terms of the supply voltage level, the internal organization of the processor and the input bus width etc. [14]. In general, the RT-level energy estimation can be classified into two types. The first type is to use a simulator to extract actual parameters for macro-model equation. The second type is based on the static analysis of the structure of the circuit so that parameters for macro-model equation could be extracted.

4.5 Gate-Level

Gate-level describes the operation of the circuit in terms of a structural interconnection of boolean logic gates such as nor, add, and xor etc., [14]. The design of gate-level is represented as a netlist, which describes the connectivity of an electronic design. Gate-level simulation (GLS) is often used at the early design stage. The simulation is performed to measure switching activity of every net of circuit. In order to perform simulation, technological information which provides timing, power information of the processor and Gate-level netlist that contains all of the logic and delays of the final system should be provided to the simulator. The drawbacks of this method are that the simulation is time consuming and Gate-level netlist is confidential for most of manufacturers.

4.6 Transistor-Level

As the name suggests, transistor level describes the operating behavior of circuit elements such as transistor, capacitors, inductors and resistors. With the rapid progress of semiconductor technology, the transistor count on a single chip has reached 2,000,000,000 in latest ARM and 1,400,000,000 latest Intel i7 [25]. In this level, transistor is often modeled as a device with only two states, “on” and “off”. In “off” state the transistor is modeled as an open circuit, while in the “on” state, the transistor is modeled by a linear resistance [14]. Transistor level power estimation employs simulation to track the current drawn from the power supply. Only a few studies have been made at this level, because the simulation is time consuming and it requires the knowledge of the circuit layout.

Chapter 5

Energy Consumption Estimation

In the previous chapter, we have presented the factors causing energy consumption on a processor. We also have compared pros and cons of energy consumption model from different abstraction levels. In this chapter, we mainly concentrate on principle and flow of energy consumption estimation at instruction-level. Section 5.1 reviews the previous researches about energy consumption estimation, special focus is given to instruction-level. Section 5.2 introduces the details of why and how to build an energy model in OVP. Section 5.3 presents the energy models and its derivation process, which is used to estimate energy consumption in an embedded processor. Having the energy model, a statistical-based energy estimation flow is given in Section 5.4. Energy consumption estimation using base cost is demonstrated in Section 5.5. Finally, we show how to estimate energy consumption for a parallelized application task on the Epiphany architecture.

5.1 Literature Review of Instruction-Level Energy Estimation

Estimation of energy\power consumption of a processor at instruction-level has been studied extensively since the growing concerns of energy efficiency especially in embedded system area. The original intention of energy consumption estimation of instruction-level is to minimize the energy cost of the software during the compilation phase so that the whole system could achieve the energy efficiency. Thus, energy consumption estimation is often considered to be the first step towards software energy minimization [26].

Table 5.1 summarizes previous works. As can be seen from Table 5.1, the instruction level energy consumption estimation is mainstream approach nowadays. The advantages of this level are relatively high performance, accuracy, and re-targetability. Thus, our discussion focuses on instruction-level since it has such advantages. The

following subsections explore the instruction-level energy consumption from four aspects.

Approach	**Target processor**	**Use of simulator**	**Claimed Accuracy**	**Abstraction level**	**Benchmark**	**Description**
Garcia et al. 2002 [27]	ARM920	cycle accurate simulator	Not available	Cycle-level	MPEG 4 video decoder	Inclusion of a power model calibrate in gate-level activity in a Cycle-Accurate simulator
Ye W et.al 2000. [28]	in-order 5-stage pipelined datapath	Simple power	Average error within 15%	Cycle-level	Benchmark Set in table 2 of this paper	Authors admitted there are many limitations.
Tajana imuni Rosing et.al 1999. [29]	ARM 710A Theoretically,it could be applied for whole ARM family	ARMulator	Simulation is found to be within 5% of the hardware measurements.	Cycle-level	Dhrystone Benchmarks	Energy Model based on the circuit-level.
Kalla et al. 2003 [30]	SPARC (scalable processor architecture)	Not have	Energy less than 5% and per-cycle power inside 15% of error	RT-level	Sort algorithms	Model based in Active and stall consumption for each individual module of the architecture.
Bazzaz et.al.2013. [31]	ARM7TDMI	SimpleScalar	Less than 6%	Instruction-level	8 Mibench benchmarks	ISS model calibrated from actual measures. Complete model with static, inter-instruction, and pipeline
Sultan et all.2009. [32]	LEON3	Not Have	Not available	Instruction-level	Not available	Propose of an instruction level power model profiling each instruction in different stages of a pipelined processor
Felipe Rosa et.al.2014. [33]	Plasma processor	OVP	Below 8%	Instruction-level	19 benchmarks from WCET	Energy consumption information are derived from gate-level simulation instead of measurement
Vivek Tiwari et.al. 1994. [26]	486DX2 and SPARClite 934	Not have	Not available	Instruction-level	Self-defined instructions instead of benchmark	It gives advice how to optimize software in order to reduce energy consumption.
Lee, Donghoon, et al. 2006. [34]	M32R-II and SH3-DSP	Gate-level simulator	Average 3% and worst case 16%	Instruction-level	JPEG and MPEG2 encoders, Compress,FFT and DCT	Training benchmarks are used in conjunction with a gate level simulator and liner optimization to generate several parameters
Lee, Sheayun, et al.2001. [35]	ARM7TDMI	Not have	Error is at most 6.33%, average is 2.5%	Instruction-level	Not available	Empirical approach and a statistical analysis technique.
Lee, Sheayun,et al. 2002. [36]	ARM7TDMI, Author claims that approach has retargetability	Not have	Error was less than 1%	Instruction-level	Not available	Compare estimated and measured energy in each clock cycle & Upgraded version of study [35], design and use a novel cycle-level energy measurement hardware.
Delicia, G et al.2014. [8]	MicroBlaze from Xilinx platform	OVP	Relative error 5% to 7.42% average	Instruction-level	6 applications	First attempt to use OVP for energy estimation

To be continue

Elikin Garcia, et al. 2011. [37]	Many-core architecture (IBM Cyclops-64)	Not have	Average error is 7.3%	Instruction-level	Matrix Multiplication	Energy consumption model for many core architectures with software managed memory hierarchy
Nikolaidis, Spiridon, et al.2005. [38]	ARM7TDMI	Not have	Error was less than 5%	Instruction-level	self-defined instructions	Energy costs are modeled in relation to a reference instruction(e.g., NOP)
Zaccaria, et al. 2003. [14]	K-issue VLIW processor	Not have	Maximum power error is 10%, average power error is 4.8%	Instruction-level	Gauss, FIR, OPT FIR, DCT IDCT, OPT DCT	propose of an instruction level energy model for VLIW pipelined architectures
Shao, Yakun Sophia and Brooks, David 2013. [39]	Intel Xeon Phi	Not have	Between 1% to 5%	Instruction-level	Not available	presents energy per instruction characterization results of Xeon Phi for different instruction types with different number of cores and threads per core configurations
Blume, Holger, et al. 2007. [24]	ARM940T and OMAP5912	ARM simulator	maximum error of 9% for the ARM940T and less than 4% for the OMAP5912	Hybrid functional-level includes instruction-level	Not available	Not only consider the energy consumed per instruction, but also the energy consumed due to cache miss hit.
Castillo, Juan, et al. 2007. [40]	ARM9TDMI and ARM TRM	Not have	Less than 11%	System-level	Bubble sort, FIR, Array, Fibonacci and Quicksort	An online analysis of the source-code without requiring simulation or even compilation. Based in the mean energy per instruction calculate from values provide by AMR manual.
Callou, Gustavo, et al. 2011. [41]	ARM7TDMI-S	Not have	7% in average	System-level	5 applications	Stochastic approach based on colored petri nets and source code analysis.

Table 5.1: State-of-Art in energy consumption estimation approaches, ordered from low abstraction level to high abstraction level

5.1.1 The method of obtaining energy consumption data

Before estimation, it needs to obtain the energy consumption in the target processor. Previous studies are based on direct measurement of energy [26], [31], [35], [36], [38], [39], energy cost information from manufacturer manual [40], gate-level, or circuit-level simulation [33]. Then, the retrieved energy values are used to characterize the energy costs of instructions for instruction-level energy estimation.

Measurement-based approach is performed by measuring the average current drawn in processor. Each instruction is put in infinite loops and current readings are taken [26]. In addition, the loops which are used to determine the energy per instruction have to satisfy certain size constraints [26]. Because on the one hand, having the same instruction in a loop would reduce the impact of the branch instruction, but on the other hand, too large loop may cause cache misses. Thus, the appropriate loop size is important for this kind of measurement. In this approach, the standard

instrumenting equipment has been used to measure the energy consumed on the processor. However, these methods often suffer due to the limitation of precision for the energy consumption. Hence, Mehta et al. [42] present that high-bandwidth digitized oscilloscope measures instantaneous power instead of the average current by capturing the voltage envelope [36]. Instantaneous current is also measured in [31], which also use a digitized oscilloscope to read the voltage difference over a precision resistor that is inserted between the power supply and the core supply pin of the processor. Nevertheless, digitized oscilloscope sometimes shows erroneous results due to current spikes. Lee, Sheayun, et al. [36] designed a cycle-level energy measurement hardware to overcome the drawback. The device can synchronize measurement data and processor clock cycle. Therefore, instruction-level energy consumption can be modeled at each clock cycle on the target processor.

Studies [8], [33] estimate energy consumption by a given application using gate-level or circuit-level simulation. Nevertheless, such hardware implementation details for off-the-shelf processors are often confidential to manufacturers. Datasheet method encounters the same problem as gate or circuit level simulation as well. Manufacturers of processor usually provide the average power per *MHz*. Yet, these values are insufficient to estimate the energy consumption accurately. Thus, on the basis of above-mentioned analysis, we believe that measurement-based method is the best way to construct energy consumption estimation model for the third party.

5.1.2 Energy consumption model

The method of constructing an energy consumption model in previous studies can be classified into two groups, statistical or non-statistical method. Note that energy consumption model that we discuss here are mainly instruction level energy model.

Non-statistical method

Prior researches mainly focus on the non-statistical energy consumption model. Tiwari et al. [26] present a seminal approach and introduce how to measure the energy consumption for each instruction. They also first recognized the inter-effect for instruction pairs. Their energy consumption approach is to prepare an energy cost table which contains the base energy cost of instructions and inter-effect for instruction pairs. The drawback of this method is also evident. Let us assume that the different type of instructions of the target processor is N, so there are $\binom{N}{2}$ ways of selecting instruction pairs from N instruction types. For example, there are more than 50 type of instructions in ARMv6M ISA [43], which means the cost table need to contain 1225 instruction pairs ($\binom{N}{2}$=1225). Moreover, different ordering of an instruction pair may yield different inter-instruction effect. Thus, measuring work for

one particular ISA would be quite heavy.

Nevertheless, the instruction set can be classified into a number of instruction groups, such as branch instructions, arithmetic logical unit instructions, and *load/-store* instructions, hence these grouped instructions could simplify the energy cost table. In [44], this technique is proposed so that energy consumption estimation can be obtained from a simple framework. In addition, non-statistical energy models include some methods which are not at instruction level. In paper [45], a high abstraction energy model is presented, the model does not consider any details of inner design. And the energy consumption model is based on total cycles, stall cycles from hardware events counters. The total energy consumption on the target processor can be calculated in this way:

$$p_{cpu} = \frac{w * p_{work} + (c - w) * p_{stall}}{c} \tag{5.1}$$

$$E_{cpu} = T_n * p_{cpu} \tag{5.2}$$

where w in Equation 5.1 is work cycles of the executed program, c is the total cycles incurred by program, T_n is the total wall clock time, p_{work} and p_{stall} are independent of workload, which can be measured once and then used as constants in the model. Thus, the total energy consumption can be calculated by Equation 5.2. Therefore, this approach can be viewed as a measurement-based method and also instruction-level energy model with only one type of instruction. However, since this approach relies heavily on hardware performance counters, the estimation must be performed by means of hardware. So this approach cannot be used to estimate non-existent hardware.

Another instruction level energy consumption model presented in [44] also does not distinguish instructions. The energy consumed by a program in this method is equal to the average energy consumed per instruction, which multiplies the number of the instructions of the program. It should be noted that this method is feasible only because the target processor is CISC architecture. Since the ISA of CISC architecture heavily use caches and read only memory, the variance of energy consumed per instruction is reduced because the energy consumed for memory operation accounts for most of the total energy consumption. However, for RISC or VLIW architectures, these effects are not evident. Therefore, this method is not applicable for these kinds of processors.

Statistical method

Recently, attempts have been made to construct an energy consumption model by means of the statistical method. Multiple linear regression is often employed to

obtain coefficients for energy consumption model. These methods treat the processor as a black box without the design details of the processor. In other words, the statistical method finds the correlation between instructions and energy dissipation. The essence of the statistical method is quite simple, namely, instruction executed on processor necessarily cause energy consumed on the target processor. The statistical method is merely responsible for deriving the weight (regression coefficients) for each instruction or component.

The simplest statistical energy consumption estimation model can be expressed by [27]:

$$E_T(\lambda) = e_o * t + \sum_{i=1}^{m} e_i * N(c_i) \tag{5.3}$$

where e_o is the static power dissipated (hence $e_o * t$ is the static energy consumption), and e_i for i from 1 to M is the energy consumed by one instruction of class c_i. The e_i is the regression coefficient which can be determined by multiple linear regression. Another paper claims that the statistical model may not be particularly obvious in some circumstances [46]. For example, when writing an operand to reorder buffer, statistical model cannot determine correlation with the energy consumption factors such as instructions fetched, instructions retired, load hits, and other statistics. To overcome this problem, their energy consumption estimation model considers activity factors and architectural statistics such as macro-instructions-per-cycle (IPC), instructions-fetched-per-cycle (SIPC), number of loads, number of stores, number of branch instructions, and number of floating-point instructions. The energy consumption model is:

$$\begin{aligned} E \;=\; & c + c_1 * N(IPC) + c_1 * N(SIPC) \\ & c_2 * N(Loads) + c_3 * N(stors) + c_4 * N(FPLdSt) + \\ & c_5 * N(FP64) + c_6 * N(Branches) \end{aligned} \tag{5.4}$$

The difficulty of using this approach is to obtain IPC and SIPC. In addition, this cannot be used to estimate unknown architecture or technology [47]. Lee, Sheayun et al. [35], [36] present energy model that aims to estimate the energy consumed in each clock cycle so that all energy consumed at all pipeline stages can be summed. Their energy model can reflect the static dissipation as base cost, the number of bits fliped and the number of logical 1's as dynamic dissipation. The undetermined coefficients which are derived from linear regression characterize the energy consumption of

instructions at each stage. The energy model is given by:

$$e_s(x,y) = B_s^X + \sum_{r \in R}^{r/X} f_r^X(\beta_r(X), \beta_r(Y)) \tag{5.5}$$

where B_s^X is the base cost instruction of X at stage s, and this is an unknown coefficient. $\beta_r(X)$ and $\beta_r(Y)$ denote the binary representation of resource r defined by instruction X and instruction Y. The second term in Equation 5.5 is defined:

$$f_r^X(\beta_r(X), \beta_r(Y)) = H_s^{r/X}(h(\beta_r(X), \beta_r(Y))) + W_s^{r/X} * \omega(\beta_r(X)) \tag{5.6}$$

where $H_s^{r/X}$ and $W_s^{r/X}$ are unknown coefficients, $h(\beta_r(X), \beta_r(Y))$ represents the Hamming distance between two binary numbers i and j. $\omega(\beta_r(X))$ denotes the number of 1's of instruction X. In general, this model is essentially a combination of functional decomposition and statistical analysis method. This energy model can achieve high estimation accuracy since it takes into account energy dissipation for all instructions whenever these instructions are executed at all pipeline stages. However, their energy model and measurement device were designed for the single issue pipeline processors (ARM7TDMI). It is unknown whether or not it can be applied to the state-of-the-art processors such as *k-issue* pipeline and multiple functional unit processors.

The study [14] and related papers describe a method of instruction-based energy modeling for a class processor which has *k-issue* VLIW (Very Long Instruction Word) with *n-stage* pipeline. Since its target ISA is VLIW with *k-issue*, a very long instruction w is defined as a vector of K elements:

$$\begin{bmatrix} w^1 \\ w^2 \\ w^3 \end{bmatrix} \tag{5.7}$$

Because the energy model considers all possible combinations of operation in an instruction, the complexity of model increases exponentially with K (*k-issue*) and the number of operations in ISA. This method also classifies the ISA so that model can find the trade-off between estimation accuracy and problem complexity. For a specific program, sequential instructions of program on the target processor can be expressed as:

$$w = \langle w_1, w_2, w_3, w_4, w_5, ..w_N \rangle \tag{5.8}$$

where w_n is the *n-th* instruction which represents an instruction in Equation 5.7. Similar to Lee, Sheayun et al.'s energy model, the context switching energy dissipation (inter-instruction effects) at each pipeline stage is also considered. The energy

consumption estimation can be calculated as:

$$E(W) = \sum_{\forall n \in N} E(w_n|w_{(n-1)}) + E_c + c_o \tag{5.9}$$

where E_c is the energy consumed by the control unit. c_o is the constant energy consumption caused by start-up a sequence. In other words, these parts do not contain energy dissipation of the pipeline and they can be treated as the static energy dissipation. The term $E(w_n|w_{n-1})$ means the switching energy at the pipeline stage between the $(n-1)th$ instruction and nth instruction. Details of the term $E(w_n|w_{n-1})$ can be modeled as:

$$E(w_n|w_{n-1}) = \sum_{\forall s \in S} A_s(w_n|w_{n-1}) + I(w_n|w_{n-1}) \tag{5.10}$$

where $A_s(w_n|w_{n-1})$ is the average energy consumed per stage s when instruction w_{n-1} executed after w_n. The term $I(w_n w_{n-1})$ is the energy consumed by pipeline connections. The reason why we classify this method into the statistical method is because the term $A_s(w_n|w_{n-1})$ can be obtained by a regression model. The details of this term are given by :

$$A_s(w_n|w_{n-1}) = U_s(w_n|w_{n-1}) + \sigma_s^n + \mu_s^n \tag{5.11}$$

where $U_s(w_n|w_{n-1})$ is the ideal energy consumed without any data or structure hazard of instruction w_n at pipeline stage s. Terms σ_s^n and μ_s^n represent the energy dissipation caused by data and instruction cache miss hit respectively. Although the terms in Equation 5.11 still reflect the pipeline features, these are obtained as regression coefficients by linear regression. Therefore, this energy model can be classified into statistical method. Furthermore, switching activities ($E(w_n|w_{n-1})$ in Equation 5.10) are essentially inter-instruction effects defined in [26]. Figure 5.1 illustrates the comparison between energy consumption estimation model in [26] and [14], [35], [36], [48]. The red mark in Figure 5.1 represents method in [26], which means this method ignores pipeline details and the effective presence of other instructions. While the blue mark means pipeline-aware method such as [14], [35], [36], [48]. These models accumulate energy consumed at each clock cycle and the energy dissipation of switching activity from the previous cycle to present cycle. Therefore, relative to the method in paper [26], these models can be defined as pipeline-aware, cycle-based, instruction-level energy consumption estimation model. Although, these features make the energy model to achieve the high accuracy. Meanwhile, these models rely on the specific cycle-accurate energy measurement device and inevitably increase the complexity of the energy model.

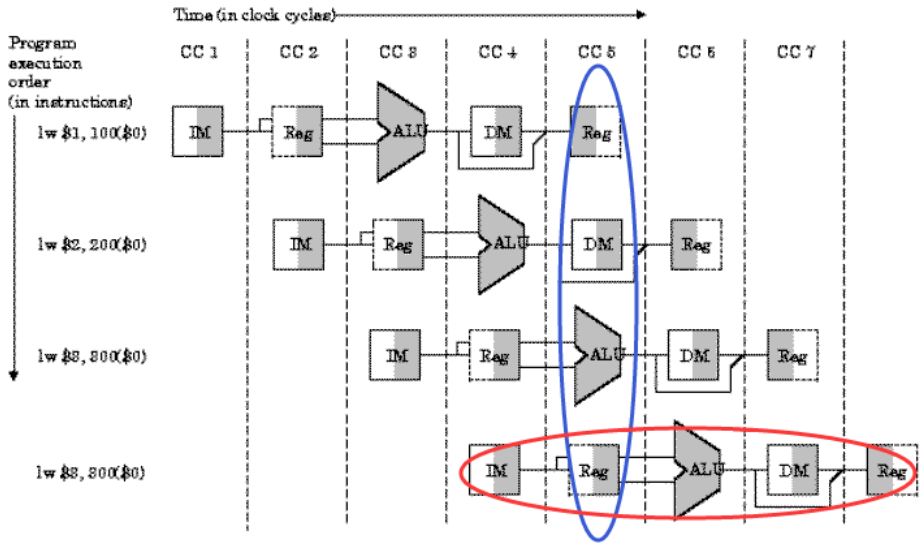


Figure 5.1: Comparison among instruction-level energy model, original source of [49]

There is another significant energy model presented in [38], which is also an improvement of study [26]. The energy model is constructed in relation to NOP instruction using mathematical derivation. For instance, the inter-instruction effects between two instructions in a 3-stage pipeline operation is calculated as:

$$E_{Instr1,Instr2} = E_{Instr1,Instr2} - E_{Instr,NOP} - E_{NOP,Instr2} - E_{NOP,NOP} \tag{5.12}$$

Unfortunately, this paper does not provide a generic inter-instruction effects equation for arbitrary pipeline stages. The model of complete energy consumption for N instructions is similar to the energy model (Equation 5.3), which expresses as:

$$E = \sum_{1}^{n} E_i + \sum_{1}^{n-1} o_{i,i-1} + \sum \epsilon \tag{5.13}$$

where E_i is the base cost of the instructions, $o_{i,i-1}$ is the inter-effect cost, and ϵ is the energy consumed due to pipeline stalls. Based on the experimental results of this paper, most of inter-instruction effective costs are negative values, and the inter-instruction cost is often less than 5% of the corresponding base cost. This conclusion is similar to the conclusion presented in [26]. Besides, these experimental results show that the inter-instruction effects are unsymmetrical. For instance, let us assume two instructions pairs "*add, ldr*" and "*ldr, add*", these two instruction pairs have different inter-instruction effects. Furthermore, this study shows that the sensitive factors, such as the register number of data-processing instructions in different register addressing mode, the number of 1's of operand values for *load/store* instructions, do not affect too much energy consumption. The worst case of these

factors only cause 3% error for estimation.

5.1.3 Instructions acquisition approaches

In order to estimate software energy consumption at instruction-level, the instruction sequence of the software must be extracted. Some works such as [35], [36] use dedicated energy consumption analyzer that analyzes instructions at every clock cycle. However, such analyzer not only take much time to develop, but also increase the complexity of the entire system. Other authors [40] propose an approach that translates the source code into intermediary code. However, just like the worst case execution time analysis, such code translator must undergo control-flow analysis, loop bound analysis, and path analysis. Thus, these methods pose extra challenges. Instruction set simulators do not encounter above problems. Studies [8], [33] obtain instructions for a given application using OVP simulator. In addition, study [24] uses ARM simulator to derive instruction word, various cycle counts (core clocks, memory bus clocks, etc.). Their energy model not only can take into account the energy consumed by instructions, but also can obtain the accurate number of cache misses because of ARM simulator.

Apart from assembly codes and simulation to obtain instruction sequence, David Brooks et al. [39] present an instruction level energy model for Intel Xeon Phi coprocessor, which extracts instruction sequence through runtime performance counter. In this study, authors introduce EPI (Energy Per Instruction) characterization results of Xeon Phi for different instruction types with the different number of cores and threads per core configurations.

5.1.4 Overall energy consumption estimation flow

The overall flow of the estimation procedure can be summarized into three phases.

1. Information acquisition phase

   This phase collects energy information that incurs energy dissipation, such as executed instructions, stall analysis, cache hit ratio, and pipeline analysis. The early research [26] mainly relies on analyzing assembly code and dedicated cache simulator. As mentioned before, some researchers use a dedicated source code translator [36], [40] to obtain instruction sequence. Some researchers ignore this phase and directly use instruction code [32], [37]. Recent researches [8], [33] use instruction simulator OVP because it is an instruction accurate simulator.

2. Characterization phase

This phase mainly determines different energy parameters such as base cost per instructions, inter-instructions effects, which defines the energy consumption estimation model. Most studies [14], [35], [36], [48] use linear regression to profile energy parameters.

3. Energy evaluation phase

   In this phase, execution profiling which was obtained in the previous phase is substituted into the energy model to calculate the total energy consumption [14], [35], [36], [48]. For these studies [26], [33], [38], this phase is done by accumulating total base cost of all instructions for a given application.

5.2 Simulation Methodology of OVP

In this section, we discuss more details of the simulation methodology for software on a target processor and the detailed analysis of why OVP is suitable for instruction-level energy estimation.

OVP offers instruction accurate and Just-in-time technology to model processor, bus, memory, and cache with high performance on PC for embedded system. The ISA provided by OVP includes MIPS, ARM, OpenRISC, PowerPC and their variants by default. Customized ISA can be developed with the help of Virtual Machine Interface libraries. Our research mainly focus on using ICM library to create an instantiation of processor, bus, memory and cache to construct the platform. Particularly, *icmNewProcessor* can be used to create instantiation of processor. Likewise, *icmNewMemory*, *icmNewBus*, and *icmNewMMC* can be used to create instantiation of memory, bus, and cache respectively. It is worth noting that these APIs should be written into a *platform.c* file. Once these instantiations are created, *icmLoadProcessorMemory* is used to load an executable and linking format (ELF) file that is generated from the simulated application to run on OVP into the instantiation of memory. Based on our observations, most of standard C codes could run on a platform that is created on OVP. Some latest C standard (C11) codes cannot run on OVP.

The outputs of OVP simulation are nominal MIPS (Million Instructions Per Second), the address of final program counter, simulated instructions, simulated MIPS and the cache hit ratio. The nominal MIPS is the rate in which the processor is configured to execute instructions on OVP. It means this rate can be set with a user attribute [9]. Simulated MIPS is the rate that is the simulated speed of the host computer. Simulated time, user time, system time, and elapsed time are unmeaningful for real hardware. In other words, these timeframe values are not close to execution time in which the application runs on the target platform. Simulated

MIPS, elapsed time, system time, simulated time and user time depends on the performance of the host PC instead of the simulated hardware. The reason for this phenomenon is that OVP does not accurately model processor speed and it is not a cycle accurate simulator [50]. In addition, the cache model in OVP determines approximately the ratio of hits to miss.

As we said before, for estimating energy consumption, we need the instruction sequence. In OVP, the address of instruction is obtained from the program counter via *icmGetPC*, then the address of this instruction is sent into *icmDisassemble* as a parameter. Finally, we extract the instructions sequence of a given application by means of *icmDisassemble*. An example of a segment of instructions sequence is given in Program 1.

Program 1 The segment of ARMv6 instructions from OVP

```
e3a07000 mov     r7,#0
e10f0000 mrs     r0,CPSR
e3c000c0 bic     r0,r0,#192
e129f000 msr     CPSR_fc,r0
e3a01000 mov     r1,#0
e59f0038 ldr     r0,[pc,#56]
```

Instruction-level energy estimation for software often need to perform control-flow analysis, loop bound analysis, and path analysis. The main advantage of OVP is that it can bypass these extra challenges. This can be validated using the following analyses.

5.2.1 Loop bound analysis

A program may spend most of execution time on loops or recursive procedures. Hence, instruction level energy estimation must either analyze loop bounds or unroll loops. The number of times of recursive call also needs to extract before estimation. OVP can generate exact instruction sequence in which the processor executes. In comparison, assembly code obtained from compiler such as GCC cannot reflect all instructions executed on the processor. The following two simple C codes help us to figure out that assembly code is unsuitable for instruction level energy estimation.

Program 2 Two simple C codes

```
int main(){
    int i=0;
    while(i<500){
        i++;
    }
}
```

```
int main(){
    int i=0;
    while(i<500000){
        i++;
    }
}
```

Both of them have the same loop body; the only difference between them is the condition. However, the number of instructions executed in the right code is much more than the left code. The left code has 4716 instructions, the right has 5000217 instructions, and both of them are obtained by OVP with ARMv6 ISA. The execution time of the right code must be far longer than the left code. Thus, the energy consumption of two codes are different, no matter what kind of processors run these codes. Unfortunately, assembly code could not give us information about change of loop bound. The assembly code of above C codes is shown in Figure 5.2. Both of two codes have the same assembly code. Another example is recursive. Let

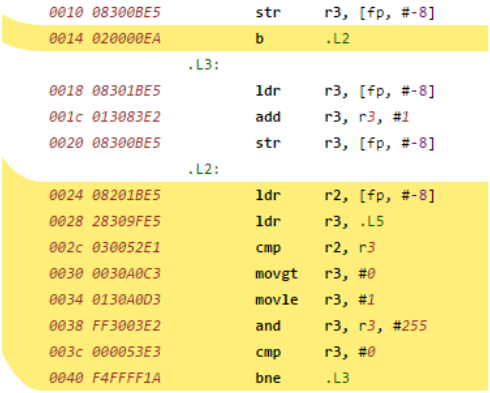


Figure 5.2: The assembly code of program 2

us consider two common Fibonacci recursive codes, as Program 3 shows:

Program 3 Two C codes of Fibonacci

```
int Fibonacci(int n){
   if ( n == 0 )
      return 0;
   else if ( n == 1 )
      return 1;
   else
      return ( Fibonacci(n-1) + Fibonacci(n-2) );
}
// Common Fibonacci function is not repeated here.
int main(){                                          int main(){
    Fibonacci(1);                                        Fibonacci(100);
}                                                    }
```

It is obvious that the right code performs much more recursive calls than the left code does. However, both of them generate almost the same assembly codes. One different instruction between two codes is shown in Figure 5.3. The left code copies value 1 to destination register r0. In contrast, the right code copies value 100 to destination register r0. The number of simulated instructions using ARMv6 ISA on OVP is 240 and 2116 respectively.

```
                        .cfi_def_cfa 11, 4
0008 0100A0E3           mov     r0, #1
000c FEFFFFEB           bl      _Z9Fibonaccii
```

Figure 5.3: Different parts of assembly code for Program 3

5.2.2 Path analysis

Path analysis determines the execution path of the program. OVP simulation does not require path analysis. Consider two C codes which are shown in Program 4.

Program 4 Two simple C codes has different branch target

```
int main(){
    int i=10;
    if(i>10){
        i++;
    }
    else{
        int k =0;
        for(;k<1000;k++)
           printf("%d\n",k);
    }
}
```

```
int main(){
    int i=100;
    if(i>10){
        i++;
    }
    else{
        int k =0;
        for(;k<1000;k++)
            printf("%d\n",k);
    }
}
```

As can be seen from Program 4, both of them have the different branch target. Thus, the number of instructions executed on the target processor varies. The number of instructions executed that are obtained from OVP is 7153 and 232 for ARMv6 ISA respectively. It is worth nothing that "*printf*" is a complex system function, which generates hundreds of instructions. In contrast, assembly codes of these C codes cannot embody this change. The only difference between the left code and the right code is that the instruction "*cmp*" in the right code compares the value in the register r3 with intermediate value 100 instead of 10.

```
0014 08301BE5           ldr     r3, [fp, #-8]
0018 0A0053E3           cmp     r3, #10
001c 030000DA           ble     .L2
```

Figure 5.4: difference between two assembly codes for Program 4

Another example also could prove that OVP does not need to perform branch target analysis. Two simple C codes shown in Program 5 are tested as well. The left code generates 224 simulated instructions on OVP and the right code generates 223 instructions on OVP. The instructions of the "*else*" part are not generated on OVP. The reason of the first code has one more instruction is because there is one more "*cmp*" instruction need to be executed. Compared with this, unless register value is analysed, assembly code cannot provide information of execution path since all assembly code of branches are generated.

Program 5 Two simple C codes have same target branch but one with additional unexecuted branch

```
int main(){
    int i=100;
    if(i>10){
        i++;
    }
    else{
        int k =0;
        for(;k<1000;k++)
           printf("%d\n",k);
    }
}
```

```
int main(){
    int i=100;
    if(i>10){
        i++;
    }
}
```

5.2.3 Control-flow analysis

Control flow determines the execution order of subroutines in an application where the subroutine is also known as method or function. It should be noted that path analysis and loop bound analysis are included in the control-flow analysis in some classification methods. Herein, the control-flow analysis only refers to determine the execution flow of subroutines. Although we have used Program 3 to illustrate the recursive procedure analysis, it also implies the invocation of a function correctly. Thus, we do not have to perform control-flow analysis on OVP.

Therefore, based on our observations, we can conscientiously draw a conclusion that OVP is an instruction accurate simulator, which provides exact instructions that run on the target processor. By using OVP, we avoid the extra burden such as loop bound analysis, path analysis, and control-flow analysis.

5.3 Instruction-Level Energy Consumption Model Derivation Process

In Chapter 3, we have explained that the energy consumed by CMOS (Complementary Metal Oxide Semiconductor) consists of three components: switching energy, short-circuits energy and leakage current. However, in a well-designed CMOS circuit, the switching energy consumption is the dominant component while the other two components account for a small portion of total energy consumption [51]. Therefore, if the switching energy consumption could be estimated, the total energy consumed by targeted processor would be estimated.

Although the state-of-the-art processors are complex systems with several functional units [26], the instruction set is relatively simple. Whenever an instruction is executed on the datapath of the target processor, the instruction is bound to incur

the switching energy dissipation. For instance, let us assume a five stage pipeline processor. When an "*add*" instruction is executed, the instruction is fetched from the instruction cache. Hence the fetch logic consumes energy. Then in the decode stage, energy is consumed in the register file when it accesses the operands [17]. Next, energy is consumed by ALU when the operation is executed. Finally, energy is consumed when the register file is accessed at write back stage. In contrast, for "*str*" instruction, memory access stage incurs energy dissipation when the memory stores a value. It is worth noting that energy is consumed if instruction "*add*" does not go through the memory access stage and instruction "*str*" does not go through the write back stage. In study [26], this energy consumption of an individual instruction is named as base cost or pure base cost. Furthermore, the energy consumed per instruction varies due to the register numbers, register values, immediate values, operand values, operand addresses and fetch addresses. These above factors are called as energy-sensitive factors [38]. For instance, in ARMv6 ISA, as can be observed in Table 5.2.

Instruction formation	Measured energy (NJ)
1: ADD Rd, Rn, Rs, ASR Rm	2.61
2: ADD Rd, Rn, Rs, ASR imm	1.60
3: ADD Rd, Rn, Rs	1.59
4: ADD Rd, Rn, Rs, RRX	1.63

Table 5.2: Comparison of measured energy costs for different variant of ADD on ARM7TDMI [38]

The difference between these four instructions is that the type of addressing model is assorted, instructions 1 and 2 contain Arithmetic Shift Right while instruction 4 contains Rotate Right with Extend instruction. The operand in instruction 1 which is shifted incurs more energy consumption than the immediate value that is shifted in instruction 2. These measurements indicate that the number of registers in one instruction for ARM7TDMI has a significant effect on base cost instead of the other factors. Therefore, according to this observation, the effect of the energy-sensitive factors can be ignored if the number of the registers is the same for particular one type of the instruction. Moreover, same type of instruction including shift operation with an additional register such as instruction 1 in Table 5.2 consumes more energy than the normal version such as instruction 3 and shifted version without an additional register such as instruction 2 and 4.

Nevertheless, the energy consumed by an instruction is not an isolated phenomenon. For a given instruction sequence, the total energy consumption is not the summation of energy consumption by each instruction in a given instruction sequence, as previously mentioned in Section 5.1. The cause of energy consumed between two instructions is switch activity that caused the number of logical 1's

(alternatively logical 0's) and the bit flips change when one instruction executed after another instruction. In study [26], this kind of energy cost is named as inter-instruction effects. Figure 5.5 illustrates the concept of inter-instruction effects:

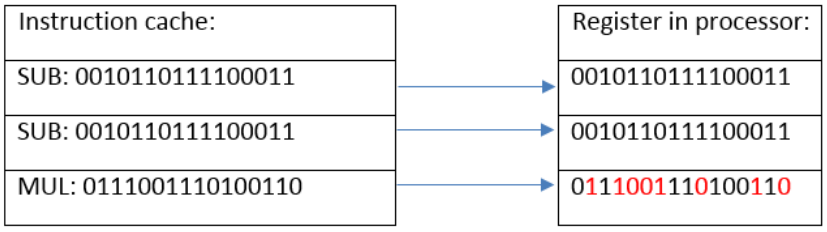


Figure 5.5: Instruction sequence with 3 instructions and their binary representations

As Figure 5.5 shows, the instruction words are transferred from the instruction cache to register in every clock cycle. The first two sub-instruction words are the same. Thus, there is no switch activity. In contrast, when the third instruction replaces the second instruction, there are 8 bits need to be switched (highlight with red in Figure 5.5). Hamming distance can be used to express the number of different bits between two instructions. The inter-instruction effects occur at every pipeline stage throughout instruction life cycle. In other words, the cost of this effect depends on the instruction itself and its previous instruction. Note that operands in two consecutive instructions incur switch activity as well. However, to simplify energy model, we ignore this factor because the energy consumption of instruction with the same number of registers has negligible deviation, it also means the value of these registers affects energy consumption negligibly.

It's worth noting that study [26] claims the cost of a pair instructions is always greater than the sum of base cost of the pair. While another study [38] claims that most of the values of the inter-instruction costs have a negative sign. However, though a paradox exists between two studies, they all reached the same conclusion that inter-instruction costs are insignificant. Study [26] presents an example that the measured cost for an instruction sequence is 332.8 mA. In contrast, the sum of base cost is 326.6 mA. While Study [38] claims that most of the inter-instruction effects costs are less than 5% of the corresponding base costs. Because the target processors of experiments are different, inter-instruction costs are either positive or negative.

Cache that is an on-chip memory may consume up to 50% of a processors total energy consumption [52], [53]. Thus, it is necessary to take into account energy consumed by the cache. The energy consumed in a cache depends on the number of cache accesses, the number of misses, cache configuration and the extent utilizing energy-efficient implementation techniques [17], [54]. For instruction-level energy

consumption estimation model, cache configuration and energy-efficient implementation are not considered since two factors are invisible at this abstraction level. Moreover, the empirical method is independent of a specific implementation. The number of cache accesses is included in the number of instructions that accesses memory (e.g., *str*, *ldr*). Furthermore, pipeline stalls also lead extra energy consumption. Instructions such as memory accessing, branch, or some multi-cycle instructions ("*mul*") incur extra pipeline stalls [31]. Although energy consumed by these pipeline stalls can be hidden in base cost of instructions, instructions cannot reflect the miss ratio and miss stalls of data and instruction cache. Hence, we should additionally take into account the misses of cache accessing.

Based on the above analysis, for a given program λ with N instructions, the energy consumption model can be expressed as:

$$E(\lambda) = \sum_{i=1}^{n} e_i * N(i) + c_m * N(misses) + \sum_{i=2}^{n} e_{i-1,i} \tag{5.14}$$

where e_i, i from 1 to n, is the energy consumed by one type instruction, $N(.)$ is a function that counts the number of a certain type of instructions in the program λ, c_m is the energy consumed by one cache misses and the $N(misses)$ is the number of times of cache miss hit. The last term is the inter-instruction cost of the instruction *i-1* and *i*. As we discussed in Section 5.1, calculating inter-instruction cost for the whole instruction set would be a heavy work even for RISC processors. Thus, to lower the complexity of the energy model, we grouped instructions based on the following principles:

- Instruction in the same group must require same computation. (e.g., integer addition *vs* integer multiplication)
- Instruction in the same group must use same functional units. (e.g., integer addition *vs* floating point addition)
- Instruction in the same group must have the same addressing mode (e.g., direct *vs* register indirect)

According with this, the ISA for any RISC architecture can be classified as:

- Branch operation: $B\langle cond\rangle$, *BL* or etc.
- Floating Point Operations (if the target processor has floating point unit):
  - Simple: *sub, add*
  - Medium: *multiply*

- Integer Arithmetic Operations:
  - Simple: *add, sub*
  - Medium: *multiply*
- Memory Operations:
  - Simple (Direct): *load immediate value, move*
  - Medium (Register): *load, str*
  - Complex (scaled register): *load, str*

There are some types of instructions that are not mentioned here. For example, Epiphany instruction set has core state, interrupt, global (chip) state and synchronization instructions [3]. Thus, this classification can be changed according to specific instruction set. Having the grouped instructions and energy model 5.14, the energy consumed for a given program λ of N instructions can be simplified as:

$$E(\lambda) = \sum_{i=1}^{n} C_{e_i} * N(i) + c_m * N(misses) + \sum_{i=2}^{n} e_{i-1,i} \tag{5.15}$$

where the changed term is only the first term in which C_{e_i} is the energy consumed per type of instructions. $N(C_i)$ is the number of each type of instructions. As shown in previous studies [26], [38], inter-instruction effects are insignificant compared with corresponding base costs. Thus, we propose an energy model that treats the inter-instruction effect as a constant.

$$E(\lambda) \simeq \sum_{i=1}^{n} C_{e_i} * N(i) + c_m * N(misses) + \llcorner \frac{N}{2} \lrcorner * \epsilon \tag{5.16}$$

where the parameter ϵ denotes average inter-instruction effects, $\llcorner \frac{N}{2} \lrcorner$ is the number of instruction pairs that is the largest integer not greater than $\frac{N}{2}$. Furthermore, for modern embedded processors, cache miss rate is not worth mentioning. Study [55] shows that the cache miss is less than 5 per 1000 instructions for a 16k cache. In addition, the cache misses caused by the energy consumption of memory accessing is not within the scope of our model. Therefore, the energy model can be further simplified as:

$$E(\lambda) \simeq \sum_{i=1}^{n} C_{e_i} * N(i) + \llcorner \frac{N}{2} \lrcorner * \epsilon \tag{5.17}$$

where ϵ represents the inter-instruction effects and also cache misses. Our energy model 5.17 is quite similar to energy model 5.3, the difference is that model 5.17 uses the length of the instruction sequence rather than the execution time.

5.4 Overall Energy Estimation Flow

In Section 5.3, the energy consumption models have been presented. In this section, we present the flow for constructing energy consumption model that uses the instruction sequence generated by OVP. Afterwards, we will show how to embed the energy consumption model into OVP.

Figure 5.6 summarizes our approach for deriving the coefficients of the energy consumption model. First, various benchmarks written in C are sent to OVP. Then, OVP generates the instruction sequence for each benchmark. After that, the statistic module counts the number of instructions in different groups. Finally, the statistical results will be used to derive the coefficients of the energy consumption model.

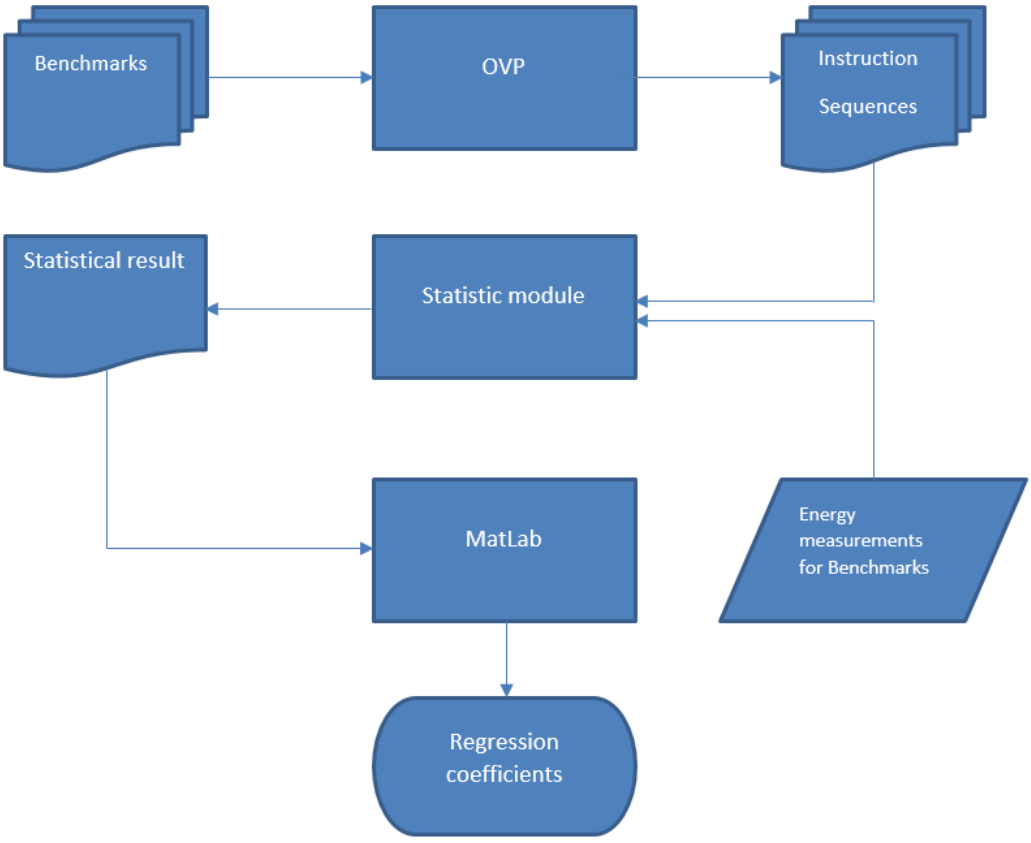


Figure 5.6: The flow of constructing energy model

The principle of Statistic Module (SM) is relatively straightforward; identify the group of the instruction, then counts the number of the instructions. Program 6 is the pseudocode of the SM. The SM requires only $O(n)$ time instead of exponential time, thus the run-time of SM increases at a near-liner trend as its input size increases. For each benchmark, the number of instructions in each instruction group and the value of the energy consumption of the target processor are obtained, so that the linear regression can derive regression coefficients (Equation 5.17) on MATLAB. Our

method is an empirical method which relies on the measurement device as previous studies.

Program 6 Algorithm of SM

```
//Input: Instruction sequences

//Output: Statistics of the number of instructions
//in each instruction group (e.g., Integer:500, Branch:20)

Dictionary(<string, int>) resultPair
For each instruction sequences
{
    If(one instruction never show up in dictionary)
    {
        resultPair.Add(this instrcion, 1);
    }
    else
    {
        The original number of this type of instructions increments;
        resultPair.Add(this type of instruction, The original value);
    }
}
```

We think that using empirical method and measurement device has the following advantages. First, for a third party, this method does not depend on internal design of processors from manufacturers. Second, the empirical method greatly simplifies the process of energy consumption estimation. Once the energy consumption model is constructed, the number of each instruction group can be substituted into the model for computing energy consumption instead of the complex implementation model. Finally, the method can be applied to a variety of different processors. It should be noted that for a different target processor, the measurement procedure should be repeated.

Once we construct the energy model, it is written into the *platform.c* code. For an example of *platform.c* code, please refer to the code A.1 of Appendix A. Whenever OVP simulates an application, the simulator verifies the correctness of the application and extracts the exact instructions executed on the target processor. After extracting instructions, the number of each instruction in one group can be substituted into the energy model to calculate energy consumption estimation. The whole simulation procedure (energy estimation procedure) is shown in Figure 5.7.

It should be noted that if one extracts instructions for an application on OVP, the simulation speed will dramatically go down. For instance, if instructions are extracted, the simulation time on *Intel i5 3210* processor is prolonged from 0.73s to 234.7s while the number of simulated instructions grows from 788,607 to 233,514,907.

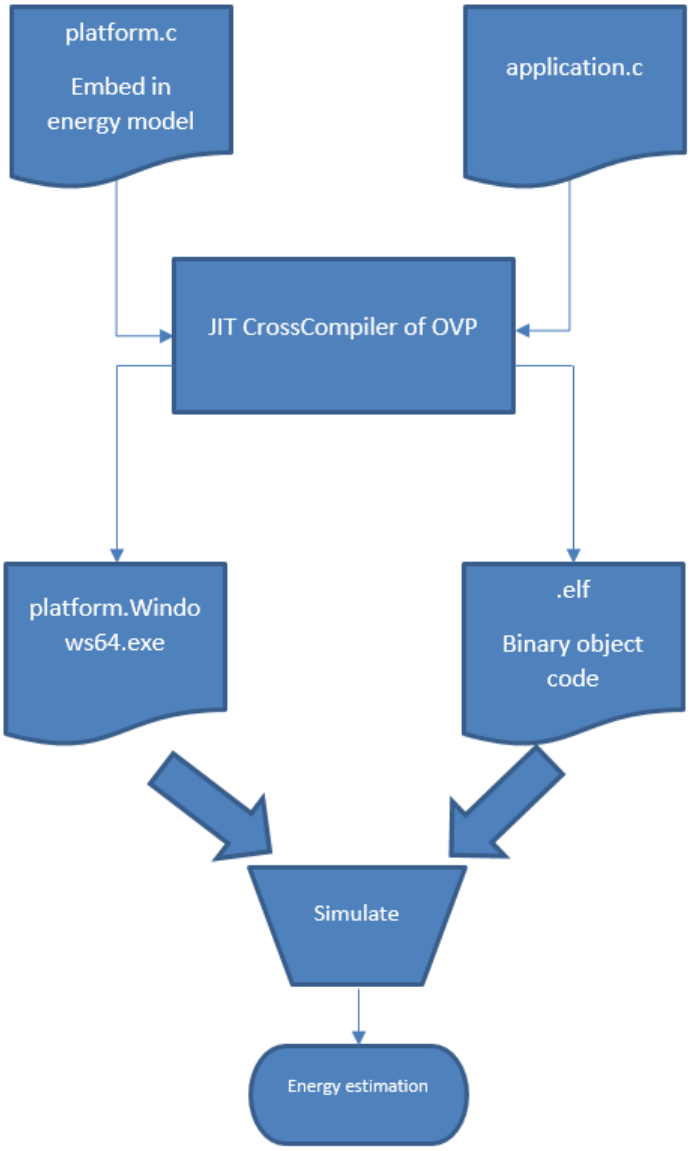


Figure 5.7: Energy estimation procedure

In contrast, if one does not extract instructions, the simulation speed does not prolong evidently even if the simulated instructions significantly increase.

5.5 Energy Consumption Estimation Using Base Cost

The total energy consumption associated with the execution of a given program consists of base energy cost, inter-instruction effects cost and cache misses as shown in Equation 5.15. Previous studies of energy estimation on OVP do not take into consideration the inter-instruction effects and cache misses. Thus, in this section, we will discuss what accuracy could be achieved if these energy dissipation parts are not available.

Previous study [38] shows that inter-instruction effects only account for 5% of the corresponding base cost. Besides, the target processor of this study is ARM7TDMI, which is a 3 stage, in-order pipelined processor. We believe this conclusion can be extended to other processors that have similar features such as single-issue, in-order execution and pipelined processor. Therefore, the energy dissipation for these classes of processors can be estimated as:

$$E = \sum_{i=1}^{N} Ce_i * N(C_i) + \sum_{i=1}^{\frac{n}{2}} Ce_{2i} * \frac{5}{100} \tag{5.18}$$

where the second term of Equation 5.18 denotes that the inter-instruction effects account for 5% of the corresponding base cost at most and we use only even order of instructions to calculate its inter-instruction effects. In order to simplify, to estimate the accuracy of energy model 5.18, we assume that the different instruction group has the same base energy cost e and it is also the most expensive base energy cost in instruction groups. Thus, the total energy consumed by N instructions on target processor can be expressed as:

$$E = N * e \pm e * \frac{5}{100} * \frac{N}{2} = N * e * (1 \pm \frac{1}{40}) \tag{5.19}$$

Hence, if we consider only the base cost of each instruction, the estimation result would be in the range (97.6% to 102.6%) of the actual energy consumption value. Moreover, if we consider the cache misses during the execution of a given instruction sequences, the total energy consumption model can be defined as follow:

$$E = N * e \pm e * \frac{5}{100} * \frac{N}{2} + e * 5 * \frac{N}{1000} = N * e * (1 \pm \frac{1}{40} + \frac{1}{200}) \tag{5.20}$$

The term $e * 5 * \frac{N}{1000}$ denotes the energy dissipation due to cache misses. The 5 is the worst case cache misses per 1000 instructions for testing applications, which is presented in study [55]. Thus, even if the cache misses occur, the sum of the base cost is still close to the total energy consumption. The sum of base cost is in

the range (97.1% to 103.1%) of the actual energy consumption, which is close to the range without consideration of the cache misses, but including inter-instruction effects. In addition, as we have mentioned in Section 5.3, the base energy cost can hide energy consumption caused by pipeline stalls except cache miss stalls. Thus, Equation 5.20 contains accuracy loss due to pipeline stalls as well. However, since our error estimation is based on the assumption that all types of instructions consuming same amount of energy, the maximum inter-instruction effects, and the worst case cache hit miss ratio. So there is a slight discrepancy between the realistic error and our estimation error.

Besides, some embedded processors employ scratchpad memory rather than traditional cache. For instance, each Epiphany core has 32K on-chip scratchpad memory. A quantitative comparison between scratchpad memory and caches that is given by study [56] shows that the reload ratio (miss ratio) for scratchpad memory and caches are very close to each other for most of test applications. Hence, we have the reason to believe that scratchpad memory miss would not affect too much on the total energy consumption in a scratchpad memory processor. Moreover, a processor that has scratchpad memory usually employs a DMA (Direct Memory Access) to mitigate pipeline stalls when the on-chip memory misses occur.

Therefore, the sum of the base cost of each instruction can achieve high accuracy without considering inter-instruction effects, cache miss ratio and pipeline stalls. However, the state-of-the-art embedded processors, fairly often, employ out-of-order execution, multi-issue, a superscalar pipeline to optimize instruction-level parallelism. Besides, the multi-threaded processor improves resource utilization, in the embedded space. These techniques improve the performance of processors. Therefore, it is worth understanding how the inter-instruction effects influence the total energy consumption of such processors. Unfortunately, to the best of our knowledge, there are no direct experimental results that indicate the relationship between base energy cost and inter-instruction effects on such processors. Study [57] indicates the inter-instruction effects are nearly a half of base costs on four-issue VLIW processor with a six-stage pipeline. On the other hand, study [26] shows that different reordering of several sequences of instructions varies only up to 2% in their current cost on Intel 486DX2-S. Although the purpose of out-of-order execution is to reduce delay and stalls by means of reordering execution that does not violate data dependencies, this technique must have more hardware such as reservation stations, common data bus, etc., and these components will incur extra energy dissipation. Tiwari et al. [58] indicate that if instruction control and data path constitute a larger fraction of silicon, the impact of inter-instruction effects should be more visible. Therefore, we propose such hypothesis that processor with above features (out-of-order execution, multi-issue, superscalar, etc.) is probably to increase inter-instruction effects cost.

Moreover, instruction effects costs would probably account for the large portion of corresponding base costs.

Regardless, the base costs are sufficient for estimating energy consumption for simple in-order pipelined processor based on the above analysis. David Brooks, et al. [39] express the same viewpoint that the inter-instruction effects within a core is negligible for the simple in-order core. But they did not give a rigorous proof or reasoning. Hence, if the target processor is Epiphany, ARM7TDMI or Intel Xeon Phi, the base cost of instructions can be summed over to estimate energy consumption. More complex processors such as ARM Cortex-A72 requires more studies to determine whether or not the summation of base costs can estimate energy consumption. Nevertheless, the energy model presented in Section 5.3 considers the processor as a black box and linear regression can be used to derive the coefficient for inter-instruction effects term in Equation 5.15. Therefore, we believe that this model could be applied to estimate processors without knowledge about inter-instruction effects.

5.6 Energy Estimation on Manycore Architecture

The total energy consumption for a manycore architecture consists of two parts, the energy consumption of each core and the energy consumption in the network-on-chip. Thus, to calculate the total energy consumption of a manycore system for given parallelized application tasks, it requires to know energy consumed by the read and write transactions that access non-local memory. In this section, we will use Epiphany architecture to interpret our energy model for manycore architectures. Proposed energy model will be the first step toward estimating energy consumption for manycore architectures.

The energy consumed for network on chip for a given application task on a network on chip can be calculated as:

$$E_{network} = N_{bytes} * N_{hop} * E_{routerperbyte} \tag{5.21}$$

where N_{bytes} and N_{hop} are the total number of bytes and the total number of hops that the network-on-chip has transferred. $E_{routerperbyte}$ denotes the energy consumed by a router for transferring one byte. Besides, the routing paths are fixed as network-on-chip is implemented with fixed routing scheme. Hence, whenever the application tasks determine the startpoint and the endpoint for data communication, we can calculate the number of hops. We can also easily extract the total number bytes of transfer in the network on chip from the source code of the application.

Moreover, for a network-on-chip of Epiphany, it consists of three dedicated 2D meshes, on-chip write (cMesh), off-chip write (xMesh), and read request (rMesh) [3].

The performance of writing is better than the performance of reading on the network on chip of Eipihany. Besides, Epiphany memory architecture is implemented with a non-cacheable on-chip memory. In order to guarantee data coherence from concurrent manycore accesses, Epiphany has to be implemented with shared memory. As mentioned before, xMesh is dedicated to deal with off-chip shared memory accessing. Therefore, we should separately consider the energy consumed on these three meshes. The total energy consumption in Epiphany network on chip can be defined as:

$$E_{network} = E_{cMesh} + E_{xMesh} + E_{rMesh} \tag{5.22}$$

Each term in Equation 5.22 can be expressed in terms of parameters in Equation 5.21.

In addition, synchronization is a critical design consideration for most parallel applications. Energy consumption model of network-on-chip must take into account energy consumed by all types of synchronization supported by Epiphany SDK such as barrier and mutex. To estimate the energy consumed by synchronization will involve the execution time estimation since the timing of reading or writing data communication need to be known. Thus, the total energy consumed by synchronization for a given application task in Epiphany architecture can be calculated as follows:

$$E_{synchronization} = \sum_{1}^{N} E_i * t \tag{5.23}$$

where E_i is the energy consumed by checking if the dependent task has finished, t is the total execution time of the dependent task, and N is the number of cores on the manycore processor.

Having Equation 5.23 and Equation 5.22, the total energy consumed by data communication can be expressed by a additive equation as follows:

$$E_{dataCommuication} = E_{synchronization} + E_{network} \tag{5.24}$$

where the term $E_{network}$ can be viewed as a static energy component that is determined before running the application task while $E_{synchronization}$ the depends on execution time of dependent tasks. Note that some simple parallelized application tasks do not have synchronization tasks. So we don't need to consider energy consumption of synchronization. Finally, the total energy consumption in the Epiphany architecture can be estimated as follows:

$$E_{Epiphany} = \sum_{1}^{N} E_{core} + E_{dataCommuication} \tag{5.25}$$

where $\sum_1^N E_{core}$ is the sum of energy consumption by each core and $E_{dataCommuication}$ is the energy consumed by data communication.

Chapter 6

Case Study

In Section 5.5, we came to the conclusion that the base cost is sufficient for estimating energy consumption for a simple in-order execution and pipelined processor. Thus, in this chapter, we use a Parallella single-board computer[a] as our experimental platform to validate our conclusion. Section 6.1 introduces the process of deriving the base cost for Epiphany and ARM Cortex M0. Section 6.2 presents and investigates the experimental result for the two processors using benchmark BEEBS. Section 6.3 represents a validation approach to the energy estimation results for Epiphany. Finally, Section 6.4 draws the conclusion based on our experiments and analysis.

6.1 The Deriving Process of Energy Base Cost

The main purpose of using the measurement device is to obtain either energy parameters (regression coefficients) or base energy cost for the energy model. For this, we have used Benchmark BEEBS that is designed to evaluate the energy consumption of embedded processors [1]. In addition to the benchmark, study [1] also presents energy measurement infrastructure, which can be used to measure most of embedded processors, and continuously monitor the current drawn, voltage and power for execution of the processor. Considering the dedicated energy consumption benchmark applications and high accuracy measurement device, we believe their measurement result is reliable. Thus, we decide to use the power dissipation of each instruction category that is presented in study [1]. Figure 6.1 shows power values of Epiphany, XMOS and ARM-Cortex M0:

[a]The Parallella is a single board computer with a dual-core ARM, FPGA and Adapteva's 16-core Epiphany coprocessor.

Category	Power (mW) Epiphany	XMOS	ARM
Integer	28	33	8.4
Floating Point	31	–	–
Memory	20	35	9.3
Branching	40	35	6.8
Other	14	–	–
Average	26	35	8.3
Average/MHz	65μW	88μW	170μW

Figure 6.1: Power dissipation for each instruction category. Figure reproduced from [1].

Since one of our research goals is to estimate energy consumption, we need to convert power value to energy value before the next step. The solution of converting power to energy can be expressed as:

$$t = \frac{c}{f} \tag{6.1}$$

$$e_{instruction} = \frac{p * t}{N} \tag{6.2}$$

where c is the cycle counts of the instruction sequence from start to finish. f is the frequency of processor under the full workload, N is the number of instructions in the instruction sequence and p is the working power.

Instruction	Cycles
LDR	2
STR	2
ADD	1
SUB	1
B	3

Table 6.1: Partial Cortex-M0 instructions and their cycle counts [59]

The cycle counts of each instruction can be found in the datasheet of a manufacturer. For example, Table 6.1 shows the timing behavior of instructions for ARM Cortex-M0 processor. For ARM Cortex M0, we convert power value to energy value using the datasheet. However, for the Epiphany core, the datasheet lacks the timing behavior of instructions. Thus, we obtained them by running individual instruction on the Parallella single-board computer. The time function "*e_ctimer_get()*" in Epiphany SDK can be used to read value of the timer in each core [10]. Table 6.2 shows the cycle count of each instruction that we obtain from experiment. We observed that lone postmodify instruction "*strd r0,[r2], r3*" takes 4294967288 cycles. Either this could occur due to the program counter overflow or may be the program counter has not been reset. Because benchmark programs that contain this

instruction take far fewer cycles compared with this instruction.

Category	Instruction	Cycles	Brief description
Integer	ADD: add r1,r7,r1	2	addition
	SUB: sub r2,r1,r0	2	subtraction
	LSL: lsl r0,r1,r2	2	logical shift left
	LSR: lsr r0,r1,r2	2	logical shift right
	ASR: asr r5,r5,0x2	2	an arithmetic shift right
	EOR: eor r2,r0,r1	2	logically XORs
	ORR: orr r2,r1,r0	2	logically ors
	AND: and r2,r1,r0	2	logically ands
	BITR: bitr r0,r0	2	reverses the order of the bits
	NOP: nop	2	do noting
Floating point	FABS: fabs r2,r1	1	calculate the absolute value
	FADD:fadd r2,r2,r0	2	addition operation for two 32-bit floating-point
	FIX: fix r2,r1	2	converts the floating-point to a fixed integer
	FLOAT: float r2,r1	2	convert the fixed-point operand to a floating-point
	FMADD: fmadd r2,r1,r0	2	multiplication and addition for floating point
	FMUL: fmul r2,r1,r0	2	multiplication for floating point
	FSUB: fsub r2,r1,r0	2	subtraction for floating point
Branching	JALR: jalr R0	2	unconditional jump
	JR: jr r0	2	unconditional jump
Memory	LDR: ldr r0,[r2,#1]	3	load data from memory to register
	STR: str r0,[r2,#0x4]	3	store data from memory to register
Others	MOV: mov r3,r1	2	copy value from between registers
	WAND: wand	2	set STATUS register and move to next instruction
	GID: gid	2	Disables all interrupts
	GIE: gie	2	Enables all interrupts in ILAT register

Table 6.2: The cycle counts of Epiphany instructions from measurement

After we get the cycle counts and power values of each instruction category, then we calculate the energy cost of each instruction category for both Epiphany core and ARM Cortex M0. Specific example of converting process is shown below:

$$e_{memory} = \frac{p * \frac{c}{f}}{N} = \frac{9.3 * 10^{-3} * \frac{2}{50} * 10^{-9}}{1} * 10^{9} = 0.000372 nanojoule \tag{6.3}$$

Note that, in Equation 6.3, 50 *MHz* is the full workload frequency of ARM Cortex M0. Besides, the energy cost for memory operation can be calculated due to the cycles of "*str*" and "*ldr*" being identical. Through the calculation one by one like Equation 6.3, we obtain the energy base cost per instruction for ARM Cortex M0 and Epiphany core, which are shown in Table 6.3.

For ARM Cortex M0, derived energy base cost per instruction agreed with the measurement value from study [60]. Since the target processors are not identical, the specific energy values are different. However, the variance of energy consumption

	Energy unit is nanojoule	
Category	**Epiphany**	**ARM Cortex-M0**
Integer	0.000093333	0.000168
Floating Point	0.00010333	-
Memory	0.0001	0.000372
Branching	0.00013333	0.000408
Other	0.000046667	-

Table 6.3: Energy per instruction for each category

for different instruction categories is more or less same. For instance, both studies indicate that memory operation consumes more energy than integer operation.

6.2 Energy Consumption of Benchmarks and Analysis

Using the energy cost per instruction for ARM Cortex M0 and Epiphany core, we can estimate energy consumption of an application using OVP and Epiphany single core simulator (ESCS). Since both Cortex M0 and Epiphany are simple in-order pipelined processors, the energy consumption estimation can ignore the inter-instruction effects and cache misses. This is decided based on the analysis and the conclusion of Section 5.5. Thus, energy consumption for benchmarks is the sum of energy per instruction. We use code that is shown in A.1 and A.2 of Appendix A to calculate the energy consumption of benchmarks. Some of the energy consumption figures of Benchmark BEEBS are given in Table B.1 of Appendix B.

Before evaluating energy consumption estimation results of the benchmarks, we are going to analyze the relationship between energy consumption and some characteristics of applications such as the number of the instructions and cycle counts. This analysis would help us understand what characteristics of applications would influence the energy dissipation on the target processors.

6.2.1 The relationship between number of instructions and energy consumption

In order to understand the relationship between number of instructions and energy consumption for ARM Cortex M0 and Epiphany core, Figures 6.2 and 6.3 are plotted:

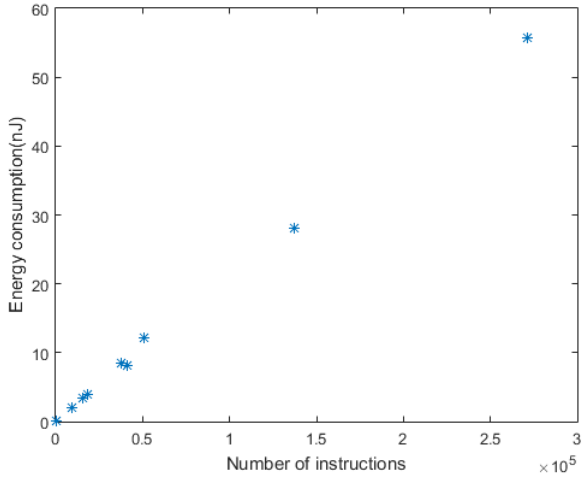


Figure 6.2: Energy consumption estimation for different number of instructions for benchmarks for ARM-Cortex M0

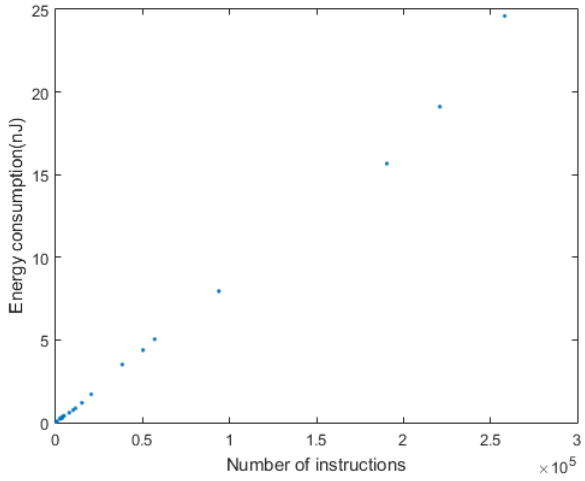


Figure 6.3: Energy consumption estimation for different number of instructions for Epiphany

As can be seen from Figure 6.2 and 6.3, the relationship between estimated energy consumption and the number of instructions are almost linear for both processors. One of the reasons is that the value of the energy parameters (Energy per instructions) are too small compared to the scale of number of instructions. For instance, the branching operation is the most expensive energy category for both processors. The energy consumed by branching operation in Epiphany is 0.00013333 nJ and it is small when compare to more than 10,000 instructions for a given benchmark. Another reason is that the standard deviation of energy per instruction (EPI) for each category of Epiphany and ARM Cortex M0 are 3.1233e-05 and 1.2943e-04 respectively, which indicate that EPI for both processors are very close to each other and hence to the mean value. It means the energy model for both processors can be regarded as a linear function. Moreover, by performing curve fitting, we found

the best fit linear equation to the data points in Figure 6.3. Then we plot the data points and the best fit linear model together in Figure 6.4 to compare fitting model and data sets. After fitting data with the model, we evaluate the goodness of fit.

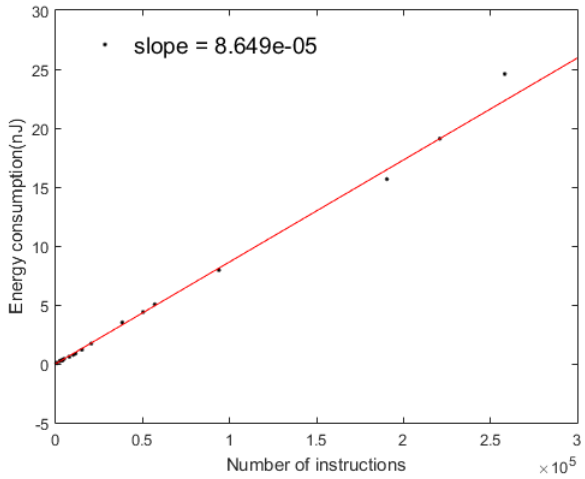


Figure 6.4: Comparing fitting model with data points from Figure 6.3.

Table 6.4 shows goodness of fit statistics of linear model. The summed square of residuals (SSE) is less than 1, it indicates that the discrepancy between the data points and the fitting model is small. The coefficient of determination (R-square) value is 0.9993, which indicates that 99.93% of the variation in the estimation energy value is captured by the best fitting model. Besides, the root mean-square deviation (RMSE) is close to 0. Thus, the differences between values calculated by the best fitting model and estimation energy values is small.

Goodness of fit	Value
SSE	0.6586
R-square	0.9993
Adjusted R-square	0.9993
RMSE	0.1692

Table 6.4: Goodness of Fit Statistics of Linear Model

Therefore, based on observations and analysis, we see that the energy consumption for Epiphany core and ARM Cortex M0 are appropriately proportional to the number of instructions.

Moreover, for most of benchmarks, energy consumption in Epiphany is less than half of ARM Cortex M0. The comparison is shown in Figure 6.5.

However, as can be seen from Figure 6.6, *fft* and *frac* consume much less energy on ARM Cortex M0. We believe this is mainly because the two benchmarks need to invoke functions in math library which is on off-chip memory, and Epiphany often generates a lot of the instructions. For instance, the total number of instructions

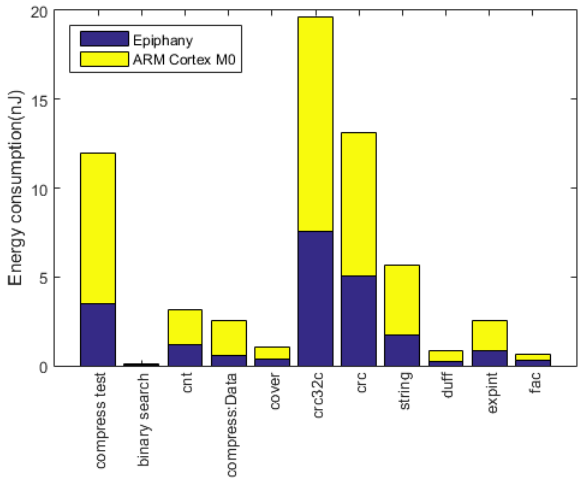


Figure 6.5: Comparison of energy consumption for both Epiphany and Cortex

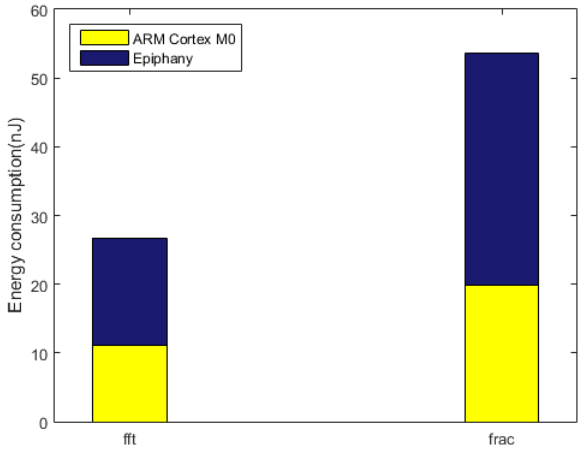


Figure 6.6: Comparison of energy consumption for both Epiphany and Cortex

of *fft* for Epiphany and ARM Cortex are 190485 and 56356 respectively. We could realize more energy efficient implementations without using such library for both benchmarks in Epiphany. Based on comparison of energy consumption for both processors, the Epiphany is more energy efficient not only in floating point operations but also in integer, memory and branching operations.

6.2.2 The relationship between cycles and energy consumption

Intuitively, more cycle counts result in more energy dissipation. As mentioned before, the execution time is one of the factors that affect the energy consumption of a processor. However, in some circumstances, energy consumption is not necessarily related to the cycle counts. As Figure 6.7 shows, in each red circle we have two benchmarks, the two benchmarks in the same red circle has similar cycle

counts. In addition to one benchmark consumes more than triple energy compared with another. For example, benchmark (*compress_test*) had generated 38267 instructions. On the other hand, benchmark (*cnt*) had generated 15108 instructions. As we mentioned before, the more instructions are generated, the more energy is consumed. Hence, two benchmarks execute similar cycle counts, but the energy consumption may not be similar.

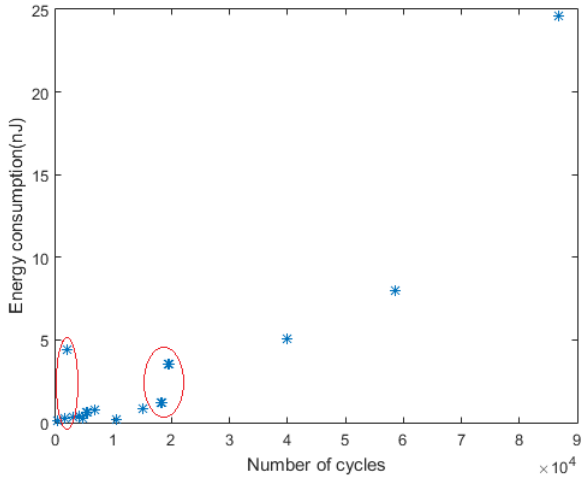


Figure 6.7: Energy consumption for different number cycle counts for Epiphany

As we mentioned earlier, the energy dissipation is proportional to the number of instructions on Epiphany core and ARM Cortex M0. Therefore, it's necessary for us to observe the relationship between cycle counts and the number of instructions. Figure 6.8 is plotted to help us understand this. As can be seen from Figure 6.8, the number of cycles doesn't seem to be proportional to the number of instructions. In other words, the relationship between the number of cycles and the number of instructions are non-linear.

In order to validate the observation, we search for the best fit of data points from Figure 6.8 by comparing graphical fit results using MATLAB. Figure 6.9 shows that there is no curve fitting for data points. Especially when we examine the goodness-of-fit statistics, we can be certain that no curve fits for the data points of Figure 6.8.

Table 6.5 shows the goodness-of-fit statistics for parametric models of curve fitting. The sum of squares due to error (SSE) for polynomial, power, sum of sine and linear fitting are huge. It indicates that the models have a big random error component and that the fit will be unuseful for prediction. Besides, root mean squared error (RMSE) is also large for these models, which means that models will be unuseful as well. Interpolating models perfectly fit in data points from Figure 6.8. However, we believe that these interpolating models are not models which we are looking for because an interpolating curve always passes through every data point [61].

Moreover, the total cycle counts for a given application is effected not only by

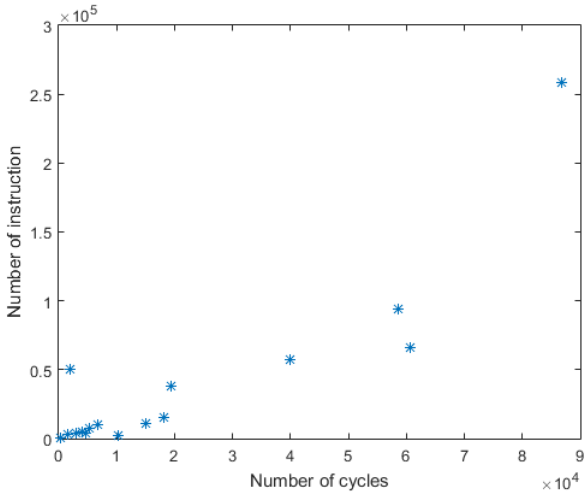
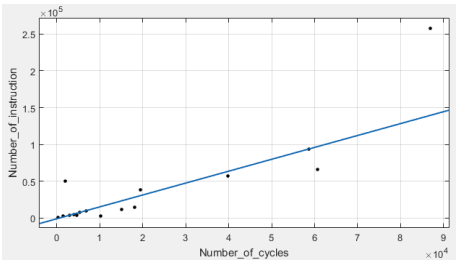
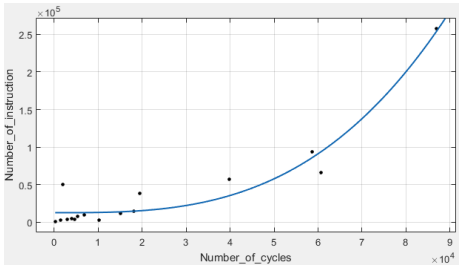


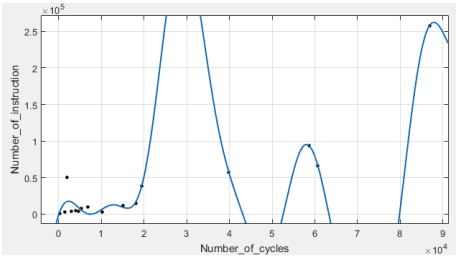
Figure 6.8: Cycle counts for different number of instructions for Epiphany



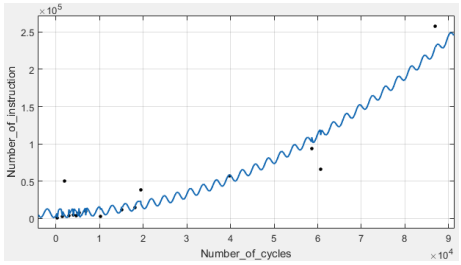
(a) Polynomial fitting with one degree



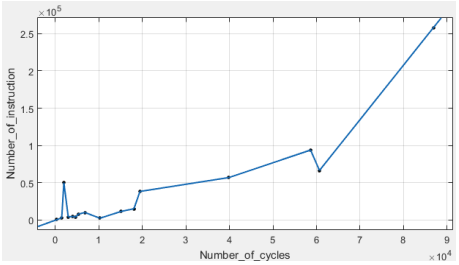
(b) Power fitting with 2 terms



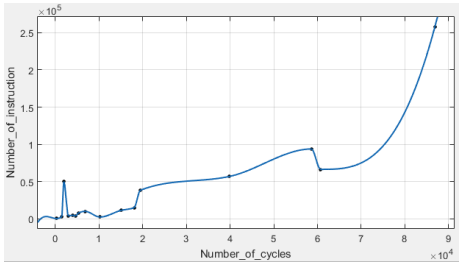
(c) Sum of sine fitting with 2 terms



(d) Linear fitting



(e) Interpolant by linear method



(f) Interpolant by shape preserving

Figure 6.9: Comparing graphical fit results

the number of instructions but also by pipeline stall. Besides branch misprediction, memory operation, cache miss hit and pipeline hazards may cause pipeline stall. Hence, pipeline stall results that the execution time is difficult to predict. Therefore, based on above analyses, only the number of instructions cannot solely determine

	Goodness of fit			
Model Types	**SSE**	**R-square**	**A R-square**	**RMSE**
Polynomial	2.818e+09	0.9554	0.9522	1.419e+04
Power	3.861e+09	0.9389	0.9295	1.723e+04
Sum of Sine	1.632e+09	0.9742	0.6128	4.04e+04
Linear Fitting	5.655e+09	0.9106	0.8968	2.086e+04
Interpolant Linear	0	1	NaN	NaN
Interpolant PCHIP	0	1	NaN	NaN
A R-square: Adjusted R-square				

Table 6.5: Goodness-of-fit statistics for parametric models

the cycle counts for a given application in Epiphany.

6.3 Validation Based on Datasheet of Epiphnay

Directly measuring the energy dissipation on the target processor is the best way to validate the proposed energy estimation approach. However, previous studies or datasheet from the manufacturer can yet be regarded as a good way to validate our approach roughly. Study [1] presents that the average power per MHz is $65\mu W$ for Epiphany core, while datasheet presented in Figure 6.10 indicates that 16 Epiphany cores (E16G301) incurs $0.7\,W$ power dissipation when the whole system is executing a heavy duty workload at $600MHz$. Thus, the power dissipation of a single epiphany

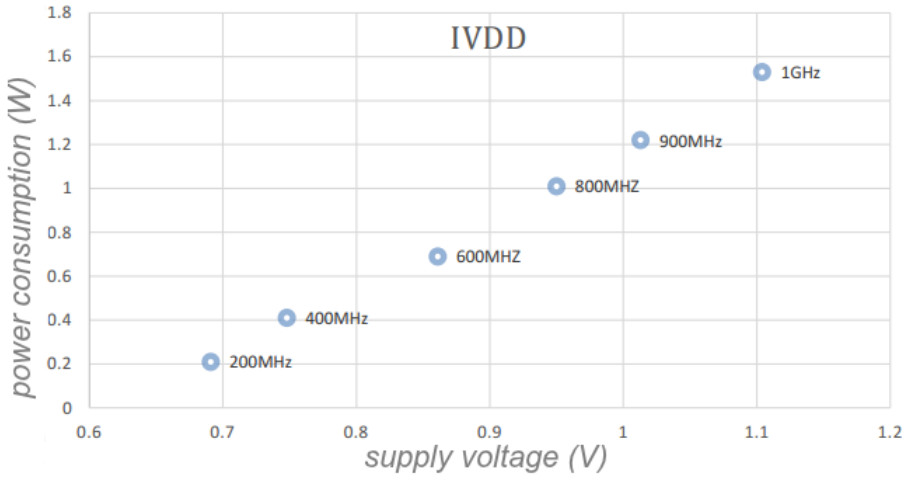


Figure 6.10: Power consumption from Addapteva's datasheet, reproduced from [62]

core is $\frac{0.7}{16}$ W. Energy dissipation is the product of power and time and the total energy consumption for a benchmark in an Epiphany core can be calculated as:

$$E = \frac{0.7}{16} * \frac{c}{600} \tag{6.4}$$

where E is energy in nanojoules, c is the cycle counts of the benchmark and 600 is the frequency of the Epiphany core under full workload. The linear Equation 6.4 is plotted in Figure 6.11 with the cycle counts and the energy consumption estimation.

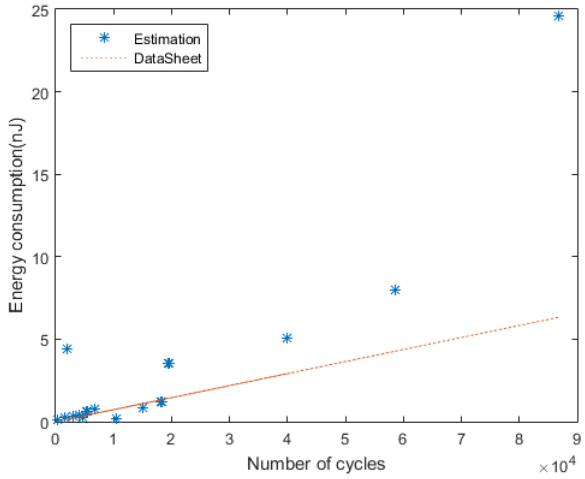


Figure 6.11: Comparison for energy consumption estimation and validation

As we can see from Figure 6.11, there are five points that have relatively big gap with the red line that is plotted by Equation 6.4. Thus, we calculate the relative change between datasheet value and estimation value for benchmarks, given by:

$$d_r = \frac{|estimation - datasheet|}{datasheet} \tag{6.5}$$

whose mean value is 2.06906, which means that the scale of difference between our estimation value and the datasheet value is on average 206.9%. Besides, the minimum of d_r is 0.09525, which means that the smallest scale of difference between our estimation value and the datasheet value is 9.525%. Furthermore, the maximum of d_r is 29.5542, which means that the biggest scale of difference is 2955.4%. Following reasons can contribute to the difference between datasheet values and estimation values for the benchmarks.

- The measurements which we used from study [1] is slightly different from Adapteva's datasheet. For example, the average power dissipation provided by the study is 65 $\mu W/MHz$. So the average power dissipation of a Epiphany core under full workload is $65 * 600 = 39000\ \mu W$. On the other hand, the value from Adapteva's datasheet is $\frac{0.7}{16} * 10^6 = 43750\ \mu W$.
- Our estimation approach itself has errors. For example, only using base cost for estimating energy consumption cannot avoid losing accuracy.

- The inherent differences between the average power dissipation and the power dissipation of an individual benchmark may cause different value.

6.4 Conclusion

Based on data and analysis, we can draw the following conclusions for energy consumption estimation in Epiphany core or ARM Cortex M0 that are simple in-order execution pipelined processors:

- The number of instructions is appropriately proportional to energy consumption on processors.
- There is indirect relationship between cycle counts and energy consumption on processors. Even if two benchmarks execute similar cycle counts, the energy consumption may be far.
- The number of instructions cannot solely determine the cycle counts. Because of pipeline stalls which can be caused by branch misprediction, cache miss hit, and pipeline hazard will lead to spending more clock cycles to complete task on processors.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

This thesis summarized previous studies on energy consumption estimation. We found that instruction-level energy estimation is the research hotspot in all abstraction levels of energy consumption estimation. Compared with energy consumption estimation approach at other abstraction levels, instruction level energy estimation has high accuracy, re-targetability, and low-complexity. Thus, we believe that future third party commercial energy estimation tools could depend on instruction-level energy consumption.

We proposed instruction-level energy consumption estimation models and presented the overall estimation flow. These models overcome complex problems such as inter-instruction effects and low-level detail. In order to extend energy consumption estimation to manycore architecture, we propose an energy model that considers all energy consumption factors in real traffic pattern of the network on chip. We have used base energy cost of Epiphany core and ARM Cortex M0 derived from the previous study in order to estimate energy consumption. Rigorous mathematical reasoning proves that energy base cost can be used to estimate energy consumption for simple pipelined, and in-order execution processors.

We have implemented the base energy cost in the OVP to estimate energy consumption for ARM Cortex M0. We also use ESCS simulator to estimate the energy consumption estimation in Epiphany core. However, we could not embed the energy consumption estimation model into the ESCS simulator. Thus, we have to use an intermediary file to save the instruction sequence. But, we could easily add the energy model into OVP as soon as OVP supports the Epiphany model. the results of base energy cost estimation for Epiphany and ARM Cortex M0 shows that the number of instructions is appropriately proportional to energy consumption. The variance between each instruction category is small. However, the results do not

mean instruction-level energy estimation is redundant. Firstly, we only use two types of processors. The variance between each instruction category to other processors could be big. Secondly, inter-instruction effects and sensitive energy effects could be more significant in more complex processors. Finally, instruction-level energy consumption estimation is the best choice for the compiler energy efficiency optimizations and instruction simulation-based energy consumption estimation. Therefore, instruction-level energy is irreplaceable in future energy estimation technological developments.

7.2 Future Work

Although the energy consumption estimation for Epiphany core has been implemented, proposed energy estimation flow has not been conducted and validated due to lack of energy measurement devices. One way to strengthen the results of the thesis could be to use measurement devices. The possible measurement solutions for future work can be classified into external measurements and on-chip monitor, as described below:

External measurements

External measurements refer to use a device that can measure current drawn or power. For example, in [26], the authors used dual-slope integrating digital ammeter to measure CPU and DRAM subsystem current. Another study [44] employs digitizing oscilloscope to measure voltage difference. Study [8] also uses oscilloscope that records power value in Comma Separated Values (CSV) file. Studies [35], [36], [38] developed a dedicated device that can monitor the current drawn in every clock cycle. They are the key device to monitor the energy consumed for instructions at given pipeline stage.

On-chip monitor

This refers to a monitor chip that gives reading of the energy consumption by the processor and external memory at microseconds rate. Texas Instrument Power monitor (TI INA231) is such kind of chip that can be embedded on the board to monitor the power consumption[63]. In addition, there are some off-the-shelf energy measurement boards that use power monitor chip. As mentioned earlier, Energy Measurement Board which was created for project MAGEEC [64] can be used to monitor energy consumed on the target processor. The current, voltage, and power data can be measured and recorded at 2,000,000 samples/second. As it can be seen in Figure 7.1, the measurement results can be viewed through energy monitoring GUI.

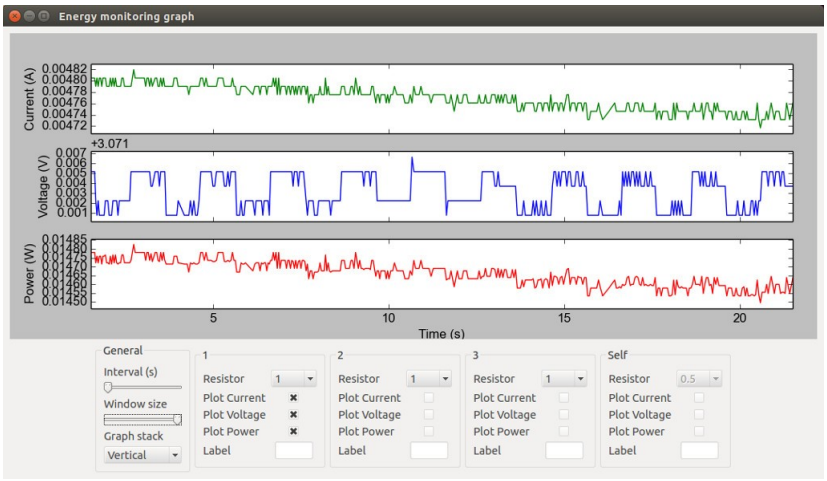


Figure 7.1: Energy monitoring GUI of MAGEEC, given by [65]

Details of future work

If we can obtain a measurement device, then we can get energy consumption model for Epiphany core by using our proposed overall estimation flow in Section 5.4. Also we could easily verify whether or not inter-instruction effects on Epiphany core is significant with corresponding base cost. Although we have concluded that inter-instruction effects do not dominate the total energy consumption on simple pipelined in-order execution processors, thus we would like to measure the energy consumption on Epiphany core so that our conclusions could be verified.

Additionally, we would like to explore the inter-instruction effects in complex embedded processor such as ARM Cortex-A9 processor which is a dual-issue superscalar, out-of-order, and dynamic length pipeline (8-11 stages) processor. Research on inter-instruction effects on such processors is still at an early stage.

Furthermore, we would like to analyze and investigate our energy estimation model for manycore architecture in Section 5.6. In order to realize this energy model, the following measuring work need to be done:

- measuring the energy consumed by a router for transferring one byte.
- measuring the energy consumed by checking if the dependent task has finished.

After measuring, we could check whether or not the energy consumption of transferring one byte in 3 different meshes of Epiphany architecture will be identical. Moreover, we will investigate the changing of energy consumption in eMesh when the data is transmitted on a different combination of three meshes on the Epiphany simultaneously. Future work will be to set up an energy consumption estimation module that can estimate the energy consumed by eCore and eMesh. This module

will employ proposed instruction-level energy consumption estimation for eCore and energy estimation model for the network on chip. In addition, this module should automatically count the total number of bytes and the total number of hops that the network-on-chip has transferred for a given application task. To estimate energy consumed by synchronization, we would study the WCET in Epiphany core. Besides, the module should be integrated into a simulator such as OVP or Epiphany simulator. Finally, though Epiphany architecture will be our target platform, we believe that the principle of energy consumption estimation is suitable for other manycore architectures.

Bibliography

[1] J. Pallister, S. Hollis, and J. Bennett, "Beebs: Open benchmarks for energy measurements on embedded platforms," *ArXiv preprint arXiv:1308.5174*, 2013.

[2] A. Sloss, D. Symes, and C. Wright, *ARM system developer's guide: Designing and optimizing system software*. Morgan Kaufmann, 2004.

[3] A. Olofsson, T. Nordström, and Z. Ul-Abdin, "Kickstarting high-performance energy-efficient manycore architectures with Epiphany," *ArXiv preprint arXiv: 1412.5538*, 2014.

[4] J. L. Hennessy and D. A. Patterson, *Computer architecture: A quantitative approach*. Elsevier, 2011.

[5] G. De Micheli and L. Benini, *Networks on chips: Technology and tools*. Academic Press, 2006.

[6] G. B. Bezerra, "Energy consumption in networks on chip: Efficiency and scaling," PhD thesis, The University of New Mexico, 2012.

[7] B. Bailey, "System level virtual prototyping becomes a reality with OVP donation from imperas," *White Paper, June*, vol. 1, 2008.

[8] G. S. P. Delicia, T. Bruckschloegl, P. Figuli, C. Tradowsky, G. M. Almeida, and J. Becker, "Bringing accuracy to open virtual platforms (OVP): A safari from high-level tools to low-level microarchitectures," in *IJCA Proceedings on International Conference on Innovations In Intelligent Instrumentation, Optimization and Electrical Sciences*, Foundation of Computer Science (FCS), 2013, pp. 22–27.

[9] I. S. Limited, *OVPsim and imperas cpumanager user guide*, English, version 2.3.15, Imperas Software Limited, Aug. 4, 2015, 160 pp.

[10] I. Adapteva, *Epiphany SDK reference*, English, version 5.13.09.10, Adapteva, Inc., Sep. 10, 2013, 164 pp., 2013-09-10.

[11] O. Jeppsson and S. A. McKee, "Towards a scalable functional simulator for the Adapteva Epiphany architecture,"

[12] O. S. Unsal, "System-level power-aware computing in complex real-time and multimedia systems," PhD thesis, University of Massachusetts Amherst, 2003.

[13] J. Srinivasan, "An overview of static power dissipation," *CiteSeer public search engine and digital libraries for scientific and academic papers in the fields of computer and information science*, pp. 1–7, 2011.

[14] V. Zaccaria, M. Sami, D. Sciuto, and C. Silvano, "Instruction-level power estimation for VLIW processor cores," *Power Estimation and Optimization Methodologies for VLIW-Based Embedded Systems*, pp. 41–65, 2003.

[15] M. Hübner and J. Becker, *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*. Springer Science & Business Media, 2010.

[16] L. Null, J. Lobur, *et al.*, *The essentials of computer organization and architecture*. Jones & Bartlett Publishers, 2014.

[17] I. Kadayif, M. Kandemir, G. Chen, N. Vijaykrishnan, M. J. Irwin, and A. Sivasubramaniam, "Compiler-directed high-level energy estimation and optimization," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 4, no. 4, pp. 819–850, 2005.

[18] C. R. Kime and M. M. Mano, *Logic and computer design fundamentals*, 2004.

[19] J. Montanaro, R. T. Witek, K. Anne, A. J. Black, E. M. Cooper, D. W. Dobberpuhl, P. M. Donahue, J. Eno, G. W. Hoeppner, D. Kruckemyer, *et al.*, "A 160-mhz, 32-b, 0.5-w CMOS RISC microprocessor," *Solid-State Circuits, IEEE Journal of*, vol. 31, no. 11, pp. 1703–1714, 1996.

[20] E. Intel, *Speed step technology for the Intel Pentium M processor*, 2004.

[21] T. Li and L. K. John, "Operating system power minimization through run-time processor resource adaptation," *Microprocessors and Microsystems*, vol. 30, no. 4, pp. 189–198, 2006.

[22] Y. Hoskote, S. Vangal, A. Singh, N. Borkar, and S. Borkar, "A 5-GHz mesh interconnect for a teraflops processor," *IEEE Micro*, no. 5, pp. 51–61, 2007.

[23] S. Hao, D. Li, W. G. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," in *Software Engineering (ICSE), 2013 35th International Conference on*, IEEE, 2013, pp. 92–101.

[24] H. Blume, D. Becker, L. Rotenberg, M. Botteck, J. Brakensiek, and T. G. Noll, "Hybrid functional-and instruction-level power modeling for embedded and heterogeneous processor architectures," *Journal of Systems Architecture*, vol. 53, no. 10, pp. 689–702, 2007.

[25] Wikipedia. (2015). Transistor count — wikipedia, the free encyclopedia, [Online]. Available: https://en.wikipedia.org/w/index.php?title=Transistor_count&oldid=672363357 (visited on 08/15/2015).

[26] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: A first step towards software power minimization," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 2, no. 4, pp. 437–445, 1994.

[27] A. B. A. García, J. Gobert, T. Dombek, H. Mehrez, and F. Petrot, "Cycle-accurate energy estimation in system level descriptions of embedded systems," in *Electronics, Circuits and Systems, 2002. 9th International Conference on*, IEEE, vol. 2, 2002, pp. 549–552.

[28] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "The design and use of simplepower: A cycle-accurate energy estimation tool," in *Proceedings of the 37th Annual Design Automation Conference*, ACM, 2000, pp. 340–345.

[29] T. Šimunić, L. Benini, and G. De Micheli, "Cycle-accurate simulation of energy consumption in embedded systems," in *Proceedings of the 36th annual ACM/IEEE Design Automation Conference*, ACM, 1999, pp. 867–872.

[30] P. Kalla, J. Henkel, and X. S. Hu, "Sea: Fast power estimation for micro-architectures," in *Proceedings of the 2003 Asia and South Pacific Design Automation Conference*, ACM, 2003, pp. 600–605.

[31] M. Bazzaz, M. Salehi, and A. Ejlali, "An accurate instruction-level energy estimation model and tool for embedded systems," *Instrumentation and Measurement, IEEE Transactions on*, vol. 62, no. 7, pp. 1927–1934, 2013.

[32] S. Sultan and S. Masud, "Rapid software power estimation of embedded pipelined processor through instruction level power model," in *Performance Evaluation of Computer & Telecommunication Systems, 2009. SPECTS 2009. International Symposium on*, IEEE, vol. 41, 2009, pp. 27–34.

[33] F. Rosa, L. Ost, T. Raupp, F. Moraes, and R. Reis, "Fast energy evaluation of embedded applications for many-core systems," in *Power and Timing Modeling, Optimization and Simulation (PATMOS), 2014 24th International Workshop on*, IEEE, 2014, pp. 1–6.

[34] D. Lee, T. Ishihara, M. Muroyama, H. Yasuura, and F. Fallah, "An energy characterization framework for software-based embedded systems," in *Embedded Systems for Real Time Multimedia, Proceedings of the 2006 IEEE/ACM/IFIP Workshop on*, IEEE, 2006, pp. 59–64.

[35] S. Lee, A. Ermedahl, S. L. Min, and N. Chang, "An accurate instruction-level energy consumption model for embedded RISC processors," in *ACM SIGPLAN Notices*, ACM, vol. 36, 2001, pp. 1–10.

[36] S. Lee, A. Ermedahl, and N. Min, "Statistical derivation of an accurate energy consumption model for embedded processors," *Submitted to special issue on Power-Aware Embedded Computing: The Role of Static, Dynamic, and Adaptive Techniques*, 2002.

[37] E. Garcia, D. Orozco, and G. Gao, "Energy efficient tiling on a many-core architecture," in *Proceedings of 4th Workshop on Programmability Issues for Heterogeneous Multicores (MULTIPROG-2011)*, 2011, pp. 53–66.

[38] S. Nikolaidis, N. Kavvadias, T. Laopoulos, L. Bisdounis, and S. Blionas, "Instruction level energy modeling for pipelined processors.," *J. Embedded Computing*, vol. 1, no. 3, pp. 317–324, 2005.

[39] Y. S. Shao and D. Brooks, "Energy characterization and instruction-level energy model of Intel's Xeon Phi processor," in *Proceedings of the International Symposium on Low Power Electronics and Design*, IEEE Press, 2013, pp. 389–394.

[40] J. Castillo, H. Posadas, E. Villar, and M. Martínez, "Energy consumption estimation technique in embedded processors with stable power consumption based on source-code operator energy figures," in *XXII Conference on Design of Circuits and Integrated Systems*, 2007.

[41] G. Callou, P. Maciel, E. Tavares, E. Andrade, B. Nogueira, C. Araujo, and P. Cunha, "Energy consumption and execution time estimation of embedded system applications," *Microprocessors and Microsystems*, vol. 35, no. 4, pp. 426–440, 2011.

[42] H. Mehta, R. M. Owens, M. J. Irwin, R. Chen, and D. Ghosh, "Techniques for low energy software," in *Proceedings of the 1997 international symposium on Low power electronics and design*, ACM, 1997, pp. 72–75.

[43] A. L. ARMv6-M, *ARMv6-M architecture reference manual.* English, version 6, ARM Ltd., Sep. 1, 2010, 436 pp.

[44] J. T. Russell and M. F. Jacome, "Software power estimation and optimization for high performance, 32-bit embedded processors," in *Computer Design: VLSI in Computers and Processors, 1998. ICCD'98. Proceedings. International Conference on*, IEEE, 1998, pp. 328–333.

[45] B. M. Tudor and Y. M. Teo, "On understanding the energy consumption of arm-based multicore servers," in *ACM SIGMETRICS Performance Evaluation Review*, ACM, vol. 41, 2013, pp. 267–278.

[46] M. D. Powell, A. Biswas, J. S. Emer, S. S. Mukherjee, B. R. Sheikh, and S. Yardi, "Camp: A technique to estimate per-structure power at run-time using a few simple parameters," in *High Performance Computer Architecture, 2009. HPCA 2009. IEEE 15th International Symposium on*, IEEE, 2009, pp. 289–300.

[47] W. J. Song, S. Yalamanchili, A. F. Rodrigues, and S. Mukhopadhyay, "Instruction-based energy estimation methodology for asymmetric manycore processor simulations," in *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*, ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2012, pp. 166–171.

[48] V. Zaccaria, *Power estimation and optimization methodologies for VLIW-based embedded systems*. Springer Science & Business Media, 2003.

[49] M. Schmalz. (2001). Organization of computer systems, [Online]. Available: `https://www.cise.ufl.edu/~mssz/CompOrg/CDA-pipe.html` (visited on 07/11/2015).

[50] I. S. Limited, *OVP guide to using processor models*, English, version 0.5, Imperas Software Limited, Jan. 19, 2015, 24 pp.

[51] S. B. Furber, *ARM system Architecture*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[52] A. Malik, B. Moyer, and D. Cermak, "A low power unified cache architecture providing power and performance flexibility," in *Low Power Electronics and Design, 2000. ISLPED'00. Proceedings of the 2000 International Symposium on*, IEEE, 2000, pp. 241–243.

[53] S. Segars, "Low power design techniques for microprocessors," *Tutorial Note of the ISSCC*, 2001.

[54] K. Ghose and M. B. Kamble, "Reducing power in superscalar processor caches using subbanking, multiple line buffers and bit-line segmentation," in *Low Power Electronics and Design, 1999. Proceedings. 1999 International Symposium on*, IEEE, 1999, pp. 70–75.

[55] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*, IEEE, 2001, pp. 3–14.

[56] I. Puau and C. Pais, "Scratchpad memories vs locked caches in hard real-time systems: A quantitative comparison," in *Design, Automation & Test in Europe Conference & Exhibition, 2007. DATE'07*, IEEE, 2007, pp. 1–6.

[57] M. Sami, D. Sciuto, C. Silvano, and V. Zaccaria, "An instruction-level energy model for embedded VLIW architectures," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 21, no. 9, pp. 998–1010, 2002.

[58] V. Tiwari, S. Malik, A. Wolfe, and M. T.-C. Lee, "Instruction level power analysis and optimization of software," in *Technologies for wireless computing*, Springer, 1996, pp. 139–154.

[59] A. Cortex, *Cortex-m0 technical reference manual*, English, version r0p0, ARM Limited., Nov. 30, 2009, 68 pp.

[60] H. Blume, D. Becker, M. Botteck, J. Brakensiek, and T. G. Noll, "Hybrid functional and instruction level power modeling for embedded processors," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Springer, 2006, pp. 216–226.

[61] I. The MathWorks. (2015). Interpolants, [Online]. Available: `http://se.mathworks.com/help/curvefit/interpolants.html` (visited on 08/24/2015).

[62] I. Adapteva, *E16g301 epiphany tm 16-core microprocessor datasheet*, English, version 14.03.11, Adapteva, Inc., Mar. 11, 2014, 164 pp., 2014-03-11.

[63] D. Abdurachmanov, P. Elmer, G. Eulisse, R. Knight, T. Niemi, J. K. Nurminen, F. Nyback, G. Pestana, Z. Ou, and K. Khan, "Techniques and tools for measuring energy efficiency of scientific software applications," in *Journal of Physics: Conference Series*, IOP Publishing, vol. 608, 2015, p. 012 032.

[64] M. Ghumran, "Analysis of energy efficient compilation," PhD thesis, University of Bristol, 2014.

[65] A. Back. (2014). Now taking orders for energy-measurement hardware, [Online]. Available: `http://mageec.org/2014/08/21/now-taking-orders-for-energy-measurement-hardware/#more-474` (visited on 08/21/2015).

Appendix A

Codes For This Project

A.1 Energy Model in OVP

Listing A.1: platform.c code

```
/*
Author  Liu Ke
2015-03-07
 */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <limits.h>
#include "icm/icmCpuManager.h"
#include "icm/icmTypes.h"
#define SIM_ATTRS 0
// this is could be changed at runtime
#define PROCESSOR_COUNT_DEFAULT 1
int bytesReading;
int bytesWriting;
int readingTimes;
int writeingTimes;
double EnergyConsumption;
long inteNumber;
long branchNumber;
long memoryNumber;
static ICM_MEM_WRITE_FN(watchWriteCB)
{
        readingTimes++;
```

```
        bytesReading += bytes;
        //icmPrintf("Detected Written 1  to 0x%08x  Bytes
            %d:\n", (Int32)address, bytes);
}


static ICM_MEM_WATCH_FN(extMemReadCB)
{
        writeingTimes++;
        bytesWriting += bytes;
        //icmPrintf("Detected Memory Reading 1 from 0x%08
           x  Bytes %d:\n", (Int32)address ,bytes);
}

void HandleInstruction (const char *p);
int main(int argc, char ** argv)
{
    Uns32 numberProcessors = PROCESSOR_COUNT_DEFAULT;
    if(argc!=2 && argc!=3)
        {
        icmPrintf("usage:␣%s␣<application␣name>␣[<number␣
           of␣processors␣-␣default␣24>]\n", argv[0]);
        return -1;
    }
    if (argc==3)
        {
        sscanf(argv[2], "%d", &numberProcessors);
    }
    // Initialize ICM
    // Require Imperas intercepts because the application
        uses impProcessorId() to get processor id
    icmInitAttrs icmAttrs = ICM_VERBOSE |
       ICM_ENABLE_IMPERAS_INTERCEPTS | ICM_STOP_ON_CTRLC;
    icmInitPlatform(ICM_VERSION, icmAttrs, 0, 0, "
       platform");

    // select library components
        const char *vlnvRoot = NULL; //When NULL use
           default library
```

```
    const char *model = icmGetVlnvString(vlnvRoot, "
       arm.ovpworld.org", "processor", "arm", "1.0",
       "model");
const char *semihosting = icmGetVlnvString(vlnvRoot,
   "arm.ovpworld.org", "semihosting", "armNewlib", "
   1.0", "model");
    icmAttrListP userAttrs = icmNewAttrList();
icmAddStringAttr(userAttrs, "variant", "ARMv6");
    icmProcessorP processor[numberProcessors];
icmBusP bus[numberProcessors];
icmMemoryP memory[numberProcessors];
char name[32];
Uns32 i;

for (i=0; i<numberProcessors; i++ )
    {
    sprintf(name, "cpu%d", i);
    processor[i] = icmNewProcessor(
        name,               // CPU name
        "arm",              // CPU type
        i,                  // CPU cpuId
        0,                  // CPU model flags
        32,                 // address bits
        model,              // model file
        "modelAttrs",       // morpher attributes
        SIM_ATTRS,          // simulation attributes.
            enable tracing etc
                userAttrs,                  //
                   user-defined attributes
        semihosting,        // semi-hosting file
        "modelAttrs"        // semi-hosting
           attributes
    );

    // create the processor busses
    sprintf(name, "bus%d", i);
    bus[i] = icmNewBus(name, 32);

    // connect the processor busses
```

```
icmConnectProcessorBusses(processor[i], bus[i],
   bus[i]);

// create memory
sprintf(name, "memory%d", i);
memory[i] = icmNewMemory(name, ICM_PRIV_RWX, 0
   xffffffff);

// connect memory
icmConnectMemoryToBus(bus[i], "mp1", memory[i],
   0x00000000);

icmPrintf("\nBUS[%d] CONNECTIONS\n",i);
icmPrintBusConnections(bus[i]);

// load the application executable file into the
   processor memory space
// (this allow the processor to see application
   symbols and sets the
// start address)
icmLoaderAttrs loadAttrs = ICM_LOAD_VERBOSE|
   ICM_SET_START;
        char *fileName = argv[1];
        char dest[50];
    strcpy(dest,fileName);
        char * pch;
    pch = strtok (dest," ,.-");
        strcat(pch, ".txt");
        icmPrintf("usage: %s\n", pch);
        FILE *fp;
        fp = fopen(pch, "wb");
if(!icmLoadProcessorMemory(processor[i], argv[1],
    loadAttrs, False, True))
        {
    icmMessage("E", "PLATFORM_LOAD", "Failed to
       load %s onto processor %d", argv[1], i);
    // terminate simulation and free simulation
       data structures
    icmTerminate();
```

```
            return -1;
        }
        icmAddWriteCallback(
        processor[i],           // processor
        0x00000000,          // low address
        0xffffffff,          // high address
        watchWriteCB,        // callback to invoke
        "watch termination" // user data passed to
           callback
    );

        icmAddReadCallback(
        processor[i],           // processor
        0x00000000,          // low address
        0xffffffff,          // high address
        extMemReadCB,        // callback to invoke
        "watch termination" // user data passed to
           callback
    );
        Bool done = False;
        while (!done)
        {

                Uns32 currentPC = (Uns32)icmGetPC(
                   processor[i]);
                const char *p = icmDisassemble(processor[
                   i], currentPC);
                HandleInstruction(p);
        done = (icmSimulate(processor[i], 1) !=
           ICM_SR_SCHED);
        }
    icmTime  simulationTime =  icmGetCurrentTime();
        icmPrintf("Number of integer instructions is [%ld
           ]\n",inteNumber);
        icmPrintf("Number of branch instructions is [%ld
           ]\n",branchNumber);
        icmPrintf("Number of memory instructions is [%ld
           ]\n",memoryNumber);
        icmPrintf("SimulationTime is [%f] s\n",(double)
```

```
            simulationTime);
        //Energy model
        EnergyConsumption = inteNumber*0.000168 +
            branchNumber*0.000408+memoryNumber*0.000372;
        icmPrintf("Total␣EnergyConsumption␣is␣[%lf]␣nJ\n"
            ,EnergyConsumption);
}
    icmTerminate();
    return 0;
}
void HandleInstruction (const char *p)
{

        if ((strstr(p, "ltr") != NULL)||(strstr(p, "str")
            != NULL))
        {
                //memory instructions
        memoryNumber++;
        }
        else if((strstr(p, "b") != NULL))
        {
                char array[50];
                strcpy(array, p);
                char * pch;
                pch = strtok(array, "␣,.-");
                pch = strtok(NULL, "␣,.-");
                char result[10];
                strcpy(result, pch);
                if(strncmp(pch, "b", strlen("b")) == 0)
                {
                        if((strstr(pch, "bi") != NULL))
                                inteNumber++;
                        else
                                branchNumber++;
                }
                else
                        inteNumber++;
        }
        else
```

```
        {
                // Other instructions
                inteNumber++;

        }
}
```

A.2 C# Code for Calculating Energy Consumption of Benchmarks in Epiphany

Listing A.2: c# code

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace EpiphanyInstructionClass
{
    class Program
    {
        public static Dictionary<string, int> resultPair
           = new Dictionary<string, int>();
        public const string pathFoder = @"E:\secondTIme\
           txt";
        public static long totalNumber = 0;
        public static int floating = 0;
        public static int interger = 0;
        public static int memory = 0;
        public static int branch = 0;
        public static int other = 0;
        public static string filename;
        static void Main(string[] args)
        {
            Stopwatch stopWatch = new Stopwatch();
            stopWatch.Start();
            ////////////////////////////
            var txtFiles = Directory.EnumerateFiles(
               pathFoder, "*.txt");
            foreach (string currentFile in txtFiles)
            {
                totalNumber = 0;
                floating = 0;
```

```
            interger = 0;
            memory = 0;
            branch = 0;
            other = 0;
            Console.WriteLine(currentFile);
            filename= String.Copy(currentFile);
            CountInstrctions(currentFile);
            Console.WriteLine("
               -------------------------------------------
               ");
        }
        ////////////////////////////
        stopWatch.Stop();
        TimeSpan ts = stopWatch.Elapsed;
        string elapsedTime = String.Format("
           {0:00}:{1:00}:{2:00}.{3:00}",
            ts.Hours, ts.Minutes, ts.Seconds,
            ts.Milliseconds / 10);
        Console.WriteLine("RunTime " + elapsedTime);

    }

    static void CountInstrctions(string path)
    {
        ReadLagerFile(path);
        Console.WriteLine("Interger : " + interger);
        Console.WriteLine("Floating point : " +
           floating);
        Console.WriteLine("Memory : " + memory);
        Console.WriteLine("Branch : " + branch);
        Console.WriteLine("Other : " + other);
        totalNumber = interger + floating + memory +
           branch + other;
        Double totalEnergy = interger * 0.000093333 +
            floating * 0.00010333 + memory * 0.0001 +
            branch * 0.00013333 + other *
           0.000046667;
        Console.WriteLine(totalEnergy+" nJ");
        Console.WriteLine("Total number of
```

```
            instructions␣:␣" + totalNumber);
        using (System.IO.StreamWriter file =
        new System.IO.StreamWriter(@"C:\Users\LiuKe\
            Desktop\WriteLines2.txt", true))
        {
            file.WriteLine(filename);
            file.WriteLine("Interger␣:␣" + interger);
            file.WriteLine("Floating␣point␣:␣" +
                floating);
            file.WriteLine("Memory␣:␣" + memory);
            file.WriteLine("Branch␣:␣" + branch);
            file.WriteLine("Other␣:␣" + other);
            file.WriteLine("Total␣Energy:␣" +
                totalEnergy + "␣nJ");
            file.WriteLine("Total␣number␣of␣
                instructions␣:␣" + totalNumber);
            file.WriteLine("
                ----------------------------------------------
                ");
        }

    }
    static void ReadLagerFile(string path)
    {
        using (FileStream fs = File.Open(path,
            FileMode.Open, FileAccess.Read, FileShare.
            ReadWrite))
        using (BufferedStream bs = new BufferedStream
            (fs))
        using (StreamReader sr = new StreamReader(bs)
            )
        {
            string line;
            while ((line = sr.ReadLine()) != null)
            {
                classification(line);
            }
        }
    }
```

```
static void classification(string line)
{

    string instructions = line.Substring(25);
    int found = instructions.IndexOf("-");
    instructions = instructions.Substring(6,
       found-7);
    if (instructions.Contains("fadd") ||
       instructions.Contains("fsub") ||
       instructions.Contains("fmul") ||
       instructions.Contains("fmadd") ||
       instructions.Contains("fmsub") ||
       instructions.Contains("fix") ||
       instructions.Contains("float") ||
       instructions.Contains("fabs"))
    {
        // floating point
        floating++;
    }
    else if (instructions.Contains("add") ||
       instructions.Contains("sub") ||
       instructions.Contains("lsl") ||
       instructions.Contains("lsr") ||
       instructions.Contains("asr") ||
       instructions.Contains("eor") ||
       instructions.Contains("orr") ||
       instructions.Contains("and") ||
       instructions.Contains("bitr") ||
       instructions.Contains("nop"))
    {
        //Interger
        interger++;
    }
    else if (instructions.Contains("ldr") ||
       instructions.Contains("str"))
    {
        //memory
        memory++;
```

```
            }
            else if (instructions.Contains("bl") ||
               instructions.Contains("jr") ||
               instructions.Contains("jalr"))
            {
                branch++;
            }
            else
            {
                other++;
            }

        }
    }
}
```

Appendix B

Tables of This Thesis

B.1 Benchmarks energy consumption estimation and their instruction category details

Processor	Benchmark	Number of instructions	Energy cost(nJ)
Cortex m0	fir2dim:Image processing for the FIR filtering	Total:15516 I:12018 B:2555 M:943	3.41226
Epiphany core	The same as above	Total:612 I:156 B:48 M:192 F:0 O: 216	0.05023986
Cortex m0	Adpcm:Adaptive Differential Pulse Code Modulation	Total:788613 I:648751 B:106345 M: 33517	163.847252
Epiphany core	The same as above	Total:902104 I:350615 B:43465 M:122551 F:28760 O: 356713	70.392734616
Cortex m0	compress test: Compress test, details are not available	Total:37721 I:27984 B:3347 M: 6390)	8.443968
Epiphany core	The same as above	Total:38267 I:13230 B:457 M:20014 F:0 O: 4566	3.510208922
Cortex m0	blowfish: An encryption algorithm which can be used as a replacement for the DES or IDEA algorithms.	Total:4692523 I:3898088 B:154836 M: 33517)	955.982700

Epiphany core	The same as above	Total:4856731 I:1858535 B:168395 M:1868946 F:61440 O: 899415	431.13094751
Cortex m0	binary search: binary search in 30 numbers	Total:330 I:238 B:52 M: 40	0.076080
Epiphany core	The same as above	Total:382 I:88 B:31 M:125 F:0 O: 138	0.03128658
Cortex m0	Bubble Sort: Bubble Sort for 100 random numbers	Total:271491 I:225452 B:20655 M: 25384	55.746024
Epiphany core	The same as above	Total:258275 I:100501 B:10225 M:130519 F:100 O: 16930	24.595664393
Cortex m0	cnt:Counts non-negative numbers in a matrix.	Total:9591 I:8083 B:722 M: 786	1.944912
Epiphany core	The same as above	Total:15108 I:6454 B:869 M:1857 F:200 O: 5728	1.191909528
Cortex m0	compress:Data compression for random data.	Total:7989 I:5248 B:768 M: 1973	1.928964
Epiphany core	The same as above	Total:7825 I:2134 B:236 M:2080 F:0 O: 3375	0.596139627
Cortex m0	cover:For testing multi- path.	Total: 2784 I:1806 B:581 M:397)	0.688140
Epiphany core	The same as above	Total:3894 I:1133 B:390 M:1530 F:0 O: 841	0.349991936
Cortex m0	crc32c:Polynomial calculation	Total: 50742 I:34794 B:8531 M: 7399)	12.078468

Epiphany core	The same as above	Total:93445 I:32818 B:6168 M:14408 F:6144 O: 33907	7.543379323
Cortex m0	crc: Cyclic Redundancy Check operation	Total:41186 I:36000 B:3109 M: 2077)	8.089116
Epiphany core	The same as above	Total:56940 I:20940 B:3186 M:21252 F:0 O: 11562	5.043946254
Cortex m0	String: String operation such as search, appending	Total: 18156 I:14006 B:2238 M: 1912)	3.977376
Epiphany core	The same as above	Total:20429 I:5578 B:307 M:8875 F:0 O: 5669	1.713599007
Cortex m0	Basic math:trigonometric function calculation(sin, cos, tan, ctan, etc.)	Total: 362403 I:311658 B:50377 M: 368)	73.049256
Epiphany core	The same as above	Total:1325143 I:653818 B:65423 M:161736 F:29304 O: 414862	108.307591258
Cortex m0	Dhrystone: no floating point operations, only integer operations	Total: 363,862,195 I:266244491 B:46410404, M:51207300)	82713.634920
Epiphany core	The same as above	Total:- I:- B:- M:- F:- O: -	-
Cortex m0	dijkstra: find the shortest path in 10*10 matrix small	Total:1404375 I:1118201 B:178505 M:107660	300.738840
Epiphany core	The same as above	Total:- I:- B:- M:- F:- O: -	-
Cortex m0	dota: convert double value to a string	Total:14068 I:10508 B:1547 M:2013)	3.145356

Epiphany core	The same as above	Total:- I:- B:- M:- F:- O: -	-
Cortex m0	duff: for testing compiler, force them emit an unstructured loop	Total:2542 I:1865, B:258 M:419	0.574452
Epiphany core	The same as above	Total:2754 I:1186 B:127 M:1282 F:0 O: 159	0.263245901
Cortex m0	edn:Simple vector multiply	Total:137161 I:114019 B:8121 M:15021	28.056372
Epiphany core	The same as above	Total:220953 I:88347 B:4018 M:73672 F:6997 O: 47919	19.107846474
Cortex m0	expint: For testing loop invariant	Total: 7212 I:5148 B:1526 M:538	1.687608
Epiphany core	The same as above	Total:11366 I:2805 B:677 M:2087 F:500 O: 5297	0.859623574
Cortex m0	fac: recursive factorial, 10 times recursion	Total:1572 I:1187 B:201 M:184)	0.349872
Epiphany core	The same as above	Total:3747 I:818 B:165 M:784 F:55 O: 1925	0.272262969
Cortex m0	fasta: memory operation of string	Total:3995734 I:3262360 B:627044, M:106330)	843.465192
Epiphany core	The same as above	Total:- I:- B:- M:- F:- O: -	-
Cortex m0	fdct: Forward Discrete Cosine Transform	Total:5176 I:4277 B:65 M:834)	1.055304

Epiphany core	The same as above	Total:5759 I:2025 B:41 M:2765 F:192 O: 736	0.525152127
Cortex m0	fft:Fast fourier transform	Total:56356 I:49561 B:6655, M:140	11.093568
Epiphany core	The same as above	Total:190485 I:91962 B:9570 M:27658 F:3216 O: 58079	15.667539419
Cortex m0	fibcall: fibonacci, 28 times loops.	Total:783 I:510 B:124 M:149	0.191700
Epiphany core	The same as above	Total:785 I:165 B:52 M:425 F:0 O: 143	0.071506486
Cortex m0	fir:Filters int data array based on passed int coefficients	Total:647331 I:488144 B:58137 M:101050	143.318688
Epiphany core	The same as above	Total:537475 I:150257 B:27709 M:302351 F:25594 O: 31564	52.071102759
Cortex m0	frac: too small and too large values for an integer fraction to represent.	Total:102997 I:92059 B:10665 M:273	19.918788
Epiphany core	The same as above	Total:412159 I:195718 B:21458 M:53427 F:9404 O: 132152	33.609495938
Cortex m0	huffbench: Huffman compression/decompression	Total:909877 I:770365 B:77415 M:62097	184.106724
Epiphany core	The same as above	Total:837992 I:293451 B:21702 M:269586 F:2500 O: 250753	69.201005094
Epiphany core	Quick sort	Total:4808 I:1625 B:550 M:1145 F:0 O:1488	0.408930121

Epiphany core	Prime	Total:66465 I:24691 B:3055 M:6596 F:1296 O:30827	4.943927572
Epiphany core	Nsichneu	Total:10018 I:1561 B:513 M:3081 F:0 O:4863	0.749132724
Epiphany core	Md5	Total:2343 I:940 B:40 M:1082 F:0 O:281	0.214379647
Epiphany core	Arcfour	Total:20205 I:15050 B:2593 M:19769 F:1024 O:11769	4.382310183
Epiphany core	Ndes	Total:93838 I:28839 B:2173 M:38190 F:0 O:24636	7.950044689
I:Integer, F:Floating point, M:Memory, B:Branching, O: Others.			

Table B.1: Benchmarks energy consumption estimation and their instruction category details

Note that some data entries are missing. In such cases, data entries require libraries that are not supported by Epiphany SDK currently.

Cease to struggle and you cease to live. ——Thomas Carlyle

PO Box 823, SE-301 18 Halmstad
Phone: +35 46 16 71 00
E-mail: registrator@hh.se
www.hh.se