Operand Value Based Modeling and Estimation of Dynamic Energy Consumption of Soft Processors in FPGA

Zaid A. M. Al-Khatib

A Thesis

in

The Department

of

Electrical and Computer Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Masters of Applied Science (Computer Engineering) at

Concordia University

Montréal, Québec, Canada

April 2013

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis proposal prepared

By: Zaid A. M. Al-Khatib

Entitled: "Operand Value Based Modeling and Estimation of Dynamic Energy Consumption of Soft Processors in FPGA"

And submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Electrical and Computer Engineering)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

_____ Chair
Dr. Rabin Raut

_____ Examiner
Dr. Otmane Ait Mohamed

_____ Examiner
Dr. Andrea Schiffauerova

_____ Supervisor
Dr. Samar Abdi

Approved by _____
Dr. W. E. Lynch, Chair
Department of Electrical and Computer Engineering

_____20_____

_____
Dr. Robin A. L. Drew
Dean, Faculty of Engineering and
Computer Science

ABSTRACT

Operand Value Based Modeling and Estimation of Dynamic Energy Consumption of Soft Processors in FPGA

Zaid A. M. Al-Khatib

This thesis presents a novel method for estimating the dynamic energy consumption of soft processors in FPGA, using an operand-value-based model. The processor energy model is created at the instruction-level, which enables fast, early and accurate energy estimation. The modeling heuristic is based on the observation that the energy required to execute instructions on an FPGA implementation of a soft processor has a strong dependence on the operand values. Our energy model contains three components: the instruction base energy, the maximum variation in the instruction energy due to input data, and the impact of one's density of the operand values during software execution. The one's density refers to the number of operand bits that are set to one. We use post-place and route processor simulations as a reference to evaluate the accuracy of our model, and that of other existing instruction-level energy models, for several benchmarks. We demonstrate that our model has only 4.7% average error and 12% worst case error compared to the reference, and is more than twice as accurate as existing instruction-level models.

Key Words: Energy modeling, Soft processors, system-level design, Power estimation

ACKNOWLEDGMENT

First and foremost, I would like to acknowledge and express my sincere gratitude for my supervisor, Dr. Samar Abdi. His insight, thoughtful guidance, support, and motivation made this wonderful learning experience possible. The experience I gained working under his supervision is invaluable. I am grateful to have had the opportunity to learn from him.

Secondly, to all my teachers. For opening my eyes to the wonders of this world. For all their help, guidance and support. I would especially like to acknowledge Mr. Nizar Jumaa. His words and encouragement helped me overcome many obstacles.

To all my fellow researchers in the embedded systems lab at Concordia University, thank you for your help and support. Thank you all for the memorable time in the lab and outside. Thanks to all those who patiently listened to my endless blabbers about energy estimation.

Last but not least, I would like to thank my family for all their sacrifice, and for making all this possible. I am forever indebted to them.

Table of Contents

List of Figures

List of Tables

List of Acronyms

ABVC	Alternating Bit Value Count
ASIC	Application Specific Integrated Circuit
CLB	Configurable Logic Block
CFG	Control Flow Graph
DSP	Digital Signal Processor
DCT	Discrete Cosine Transfer
FFT	Fast Fourier Transfer
FPGA	Field Programmable Gate Array
ISS	Instruction Set Simulator
LWI	Load Word Immediate
LMB	Local Memory Bus
LBEP	Location Based Energy Profile
LLVM	Low Level Virtual Machine
OVBM	Operand Value Based Model
RISC	Reduced Instruction Set Computing
RTL	Register Transfer Level

SRA	Shift Right Arithmetic
SRL	Shift Right Logical
SWI	Store Word Immediate
VCD	Value Change Dump
VLIW	Very Long Instruction Word
XPA	Xilinx Power Analyzer
XPE	Xilinx Power Estimator

Chapter 1

Introduction

1.1 Motivation

Recent design trends have pointed to increased use of Field Programmable Gate Array (FPGA) in embedded systems. Soft-processors, implemented in FPGA, provide a convenient computational platform. Soft-processor energy consumption is a first order metric for FPGA-based embedded system design, due to its direct impact on battery life. Therefore, early and accurate modeling of soft-processor performance and energy consumption, for a given application, is needed to perform early design space exploration with reasonable confidence. Furthermore, rapid profiling of the energy consumption required to run an application is important to guide software optimizations to minimize energy requirements.

The energy consumed by any digital circuit, including processors, consists of both static and dynamic energy. The static energy, also known as leakage energy is constant, and is determined by the physical material properties as well as the fabrication technology. It does not depend on the operation of the circuit. Therefore, it is easy to measure or estimate. However, the dynamic energy, also known as switching energy, consumed by a given transistor in a digital circuit is a function of its switching frequency, load capacitance and supply voltage, the rate of which is given in Equation (1.1). The load capacitance and supply voltage for each transistor in an FPGA are fixed. However, the switching frequency is determined by the operation of the circuit, which is governed by the executing application. Therefore, modeling the dynamic energy consumption of a soft processor

remains a major challenge due to the large number of contributing factors to the individual transistor switching rates. The focus of this research is to estimate the dynamic energy consumed by the processor. Therefore, only dynamic energy and power will be discussed in this thesis.

$$P_{dynamic} = \frac{1}{2} \cdot C \cdot V_{dd}^2 \cdot f \qquad (1.1)$$

The abstraction level of the processor energy model influences the estimation accuracy and efficiency as shown in Figure 1.1. Low-level post-place-and-route models may be used for accurate estimation, but they are extremely slow, thereby hindering the design space exploration. System-level energy models, based on power modes of the processor, suffer from accuracy issues. Instruction-level models are more accurate than system-level models, but usually require large data models, especially if they incorporate the inter-instruction energy effects. Existing instruction-level models do not take the input data of the application into account, potentially leading to large inaccuracies, especially for soft processors implemented in FPGA.

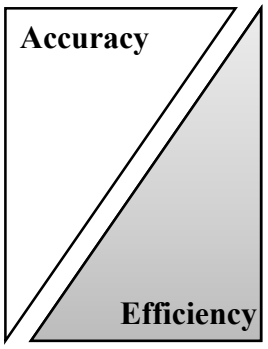
	Processor Power Model Description	Accuracy / Efficiency Abstraction
1	Gate-Level	
2	RTL-Level	
3	Pipeline state aware	
4	**Instruction-level**	
5	Analytical, instruction-class based	
6	Function-level Macro-models	
7	Mode-based	

Figure 1.1 Correlation of the processor power model abstraction level with estimation accuracy and computation efficiency [1]

Our goal, in this research, is to build an instruction-level energy model that can provide fast and accurate estimates of dynamic energy consumption for soft processors in FPGA. Our model takes into account the operand values of the instructions, for calculating the energy consumption. Previous work has shown insignificant impact of operand values on energy consumed by Application Specific Integrated Circuit (ASIC) processors [2]. However, our experimental results show that the operand values greatly affect the energy consumed by soft processor cores implemented in FPGA. Because the soft-processor data-path units are implemented on several Configurable Logic Blocks (CLBs) and Digital Signal Processors (DSPs), operand values propagate through long interconnect routes. Heuristic reasoning indicates the switching rates of these interconnects is proportional to operand value. That is because the applications are made up of small basic blocks of heterogeneous sequences of instructions which very often contain instructions with no or very small operands [3]. Such instructions include NOPs and word/byte operations. These instructions force most of the data-path signals to zero. Similarly, branch mispredictions and unfilled delay slots also reset several data-path signals to the zero state. This results in an increased probability of signals switching when instructions operating with values that contain a lot of ones (that we will refer to as one's density) are executed. Furthermore, the energy impact of some homogeneous sequences of instructions also depends on the operand value. Such constructs are often utilized by compilers to implement certain operations. For example, the Microblaze compiler uses a series of identical shift instructions to implement several operations. For such sequences of shift instructions, the energy consumed is proportional to the number of alternating bit values in the operand value.

Based on the above heuristics, an energy data model is created for the processor for a given technology. The application software is compiled for the target processor, and executed on an

instruction-set simulator or a prototype implementation to get the stream of executing instructions and the operands values metrics. The processor energy data model is used to calculate the energy of each instruction and the cumulative energy for each basic block in the application. The basic-block energy is back annotated in the application code to derive an executable energy model.

1.2 Related Work

There are a number of modeling techniques to estimate the power and energy consumption of embedded processors. They can be categorized based on the abstraction level of the model. The most accurate models simulate the processor at the gate and transistor levels. Examples of tools, based on such models, include Xilinx's X*Power Analyzer* (XPA) [4] and Synopsys's Power Compiler [5]. Although accurate, low-level simulation models suffer from very slow simulation speeds. That is because they require very large signal *Value Change Dump* (VCD) files that are generated by tedious post-place and route simulations. For instance, using top of the line, off the shelf PCs, it took an hour and 15 minutes for XPA to estimate the average power consumption of the Dhrystone benchmark [6], running on a Microblaze processor [7]. Similarly, XPA took over 10 hours of simulation to estimate the average power consumption of a JPEG encoder application [8]. These times are for the simulation running on a quad-core i7 PC with 16 GB RAM.

Estimation techniques that use high levels of abstraction typically use statistical approach to generate very rough energy estimates. Such tools are used in early design space exploration because of their speed. Xilinx introduced the Xilinx *XPower Estimator* (XPE) tool which uses highly abstracted statistical models [9]. It was developed to complement their XPA tool, providing their users power estimation tools from both ends of the abstraction spectrum. Others have also attempted to model processors power characteristics using back-propagation neural networks [10] [11]. Although this model can be used efficiently, its accuracy is directly dependent on the training

set used to generate the neural network weights. Although it might be useful to efficiently compare alternative implementations of one algorithm, this approach cannot be used to estimate the power consumption of different programs efficiently.

Other processor energy and power models typically characterize either the processor functional blocks or the workload of the processor. Functional block characterization through macro modeling is a conventional approach used for analyzing hardware RTL models, and has been applied to ASIC processors [12] [13] [14] as well as soft processors implemented in FPGA [15]. However, such techniques use signal activity rates that are derived using extensive time consuming simulations. To over-come this limitation, several works have proposed using such an approach to model a processors instruction set [16]. However, the generated models were not used to estimate the energy requirements of full applications, and hence were not validated. We have identified two types of models that characterize the workload of the processors. The first are state-based models, and the second are instruction level models.

State Based Models consider only the power state of the processor and timing information to estimate the energy consumption of the processor. An average dynamic power value is assigned to each power state. The dynamic energy consumed by executing an application is estimated by multiplying the weighted average power by the total execution time. Jouletrack implements a very state based model [2]. System level estimation tools like Softwatt [17], Wattch [18], and other state-based estimation techniques [19] [20] [21] use state based processor models because of their simplicity. Energy aware scheduling techniques using state based models have also been proposed [22]. However, state based models do not properly reflect the processor energy savings from software optimizations that do not reduce the execution time. For example, re-ordering assembly instructions to increase the number of repeating instructions greatly reduces energy consumption.

However, this effect cannot be observed by state based models. We have developed a state based model of the Microblaze processor to compare the proposed model with highly abstracted models. It is presented in detail in section 2.1.1.

Instruction level models are divided between first and second order models. First order models assign an estimated average energy value for each instruction in the processors instruction set. The energy required to execute a program is then estimated as the sum of the energy values assigned to all the instructions that execute as part of the program. Jouletrack models an ARM and a Hitachi ASIC processors using first order model in addition to the state based model [2]. However, the results in [2] demonstrate negligible increase in accuracy using first order models, compared to state based models. First order models for the Microblaze soft processor have been proposed in [23]. However, although the experimental results presented in [23] are generated using only two, very small, benchmark: Fast Fourier Transfer (FFT) and Matrix Multiplication. As we present in this paper, these models tend to perform well for certain types of application, especially small ones, but fail to estimate the energy of large applications. Section 2.1.3 summarizes the techniques used to develop first order instruction-level models.

Second order processor models incorporate inter-instruction energy effects on top of a base energy estimate for each instruction. That is, while analyzing the energy required for completing an instruction, the neighboring instructions are also accounted for. The reasoning is that neighboring instructions indicate the state of the processor before and after executing an instruction. Several processors, including an Intel 486DX2 processor [24] and a Fujitsu DSP [25], have been modeled using this approach. VLIW processors have also been modeled using second order models. Specifically, the StarCore VLIW DSP was modeled in the JouleQuest tool [26], and others in [27] [28]. Second order models have been applied to estimate system-level energy consumption [29].

However, second order models do not take the effects of input data for the application into account, which can lead to significant errors. Our processor energy models builds on second order models by incorporating effects of operand values of instructions. In section 2.1.4 we develop a second order model of the Microblaze processor using techniques suggested in the literature.

Automated energy estimation tools are required to use such models to estimate the energy required by an application. Several works proposed using open source compilers like Low Level Virtual Machine (LLVM) to trace the dynamic behavior of an application [30]. However, such tools have only been validated using very small and simple benchmarks, but not realistic applications. It is doubtful that such an approach would generate satisfactory results because the Control Flow Graphs (CFG) of applications generated using different compilers are often different.

1.3 Writing Conventions

To improve the readability of this thesis and to avoid excessive use of acronyms, we adopted two conventions in the writing of this thesis. We focus in this research on analyzing and estimating the dynamic energy consumption of processors. However, digital circuits also consumes static energy at a constant rate; i.e. the static power. Therefore, although statements like "energy consumed by the processor" usually imply both types of energy, we use it to refer solely to the dynamic energy consumption, unless we clearly specify otherwise. Secondly, the switching of internal signals in the processor, dictated by the instruction, causes the consumption of dynamic energy. Therefore, although we will frequently use expressions like "energy of instruction" or "energy of an application", we do not mean to imply the software itself consumes the energy. Instead, we use these phrases to refer to the dynamic energy consumed by the processor as it executes an instruction and the dynamic energy consumed by the processor as it executes an application.

1.4 Thesis Outline

The rest of the thesis is organized in 4 chapters as follows. Chapter 2 describes the processor energy models. It described energy models developed using the techniques commonly used in the literature, as well as the proposed model. We begin by describing the state of the arts processor energy models; i.e. Post-place and route models, state based models, and first, and second order instruction-level models. We define each model, and present its parameters and equations. We also describe the process used to model the Microblaze processor using each method. We further discuss the major weakness of each approach. In the second part of Chapter 2 describes the proposed Operand Value Based Model (OVBM) in detail, and the process used to generate the Microblaze OVBM. This part is broken into 3 subsections discussing the main parameters of an OVBM; the base energy cost of instructions, maximum energy variance due to operand values, and the impact of operand values on instruction energy.

Chapter 3 presents the automated tool we developed to use instruction-level energy models and estimate the energy required by the processor to execute any given application. The first part of the chapter describes the first phase of the tool, needed to generate all the parameters needed by the OVBM. We describe how the tool, in the first phase, analysis the source code and identifies the basic blocks. Furthermore, we describe how the tool automatically annotates the source code with instructions to trace the application execution and analysis the values of the operands of each executing instruction. In the second part of the chapter, we describe the second phase of the tool. Listing all the equations applied to use the energy model, execution trace, and operand metrics to generate detailed estimated energy reports.

In Chapter 4, we present the experimental results from using the energy models described in Chater2, and using the estimation tool described in Chapter 3. We compare the accuracy of the

estimates generated using state of the arts models, OVBM, and reference low-level estimation models in four sections. In the first section, we present the accuracy of the instruction-level models, demonstrating the higher accuracy and reliability from using the OVBM. In the second section, we demonstrate the speed advantage of using the instruction-level OVBM over the accurate post-place and route estimation method. In the third section, we describe the estimation granularity of estimates generated using the OVBM as compared to the accurate low-level estimation models. In the last section, we compare the effort needed to generate each of the state of the art instruction-level models and OVBM. Finally, we present our conclusion and future work in Chapter 5.

Chapter 2

Processor Energy Modeling

This chapter will describe state of the arts energy models implemented at different abstraction levels as well as the proposed Operand Value Based Model (OVBM). We also describe the energy models generated using each technique for a Microblaze soft processor implementation on a Xilinx Virtex5 FPGA. These models are used to validate the proposed technique and compare its accuracy, efficiency and granularity with the state of the arts methods as presented in Chapter 5.

2.1. State of the Art Models

As described in the literature review in Section 1.3, we categorize state of the art energy models by their abstraction level. The following subsections will describe low-level, post place and route models, high-level state based models, and two instruction level models of the Microblaze soft processor. We also present the Microblaze energy model generated using each method.

2.1.1 Post-Place and Route Model

The most accurate method to estimate the energy of a soft processor requires estimating the energy consumed by each transistor using the fundamental dynamic power equation (2.1). For the Xilinx Virtex5 FPGA, that is 1.1 billion transistors in a single chip [31], 55 million of which are used to implement a Microblaze processor. For each, the average switching frequency depends on the instructions the processor implementation is executing. Therefore, tedious post-place and routes simulations are needed to estimate the switching frequency for each signal, as the application is executed. Xilinx includes the iSim simulator in its suite of tools which is performs such simulations

[32]. Once a simulation is completed, the simulator produces a VCD file that contains the average switching rate of each signal. These values are used along with signal capacitance vectors and supply voltage level by Xilinx XPA tool [4] to estimate the average power consumed by the processor with great accuracy.

$$P_{dynamic} = \frac{1}{2} \cdot C \cdot V_{dd}^2 \cdot f \quad (2.1)$$

Using the average power consumed by the microblaze as it executes an application and the time it taken to complete the execution, the energy consumed by the application can be calculated using equation (2.2).

$$E_{application} = P_{dynamic} \cdot T \quad (2.2)$$

Because of its accuracy, this approach is used to characterize the energy consumption for the state based and instruction level energy models. Although physical energy measurement methods have been proposed and used before to generate instruction-level models [33], they are impractical for modeling soft processors. That is because it is not possible to measure only the dynamic power consumed by the soft processor. Instead, physical measurement include the total power (static and dynamic) consumed by the whole FPGA chip including memory, clock network, and system bus components.

2.1.2. State Based Model

In state based models, we assume the processor is consuming a constant average power while in the active state; i.e. while it is executing any application. Therefore, to model the dynamic energy using a state based model, a single average power parameter is sufficient to estimate the energy of any application using equation (2.3). The execution time of the application, denoted by T, is

required to estimate the energy required by an application. Using a performance estimation tool like Xilinx's iSim [32] and accurate behavioral models of the processor, we can estimate the execution time.

$$E_{application} = P_{average} \cdot T \quad (2.3)$$

One application is selected to evaluate the average dynamic power of the processor using the accurate post place and route model. The distribution of the operations in the application selected will directly impact the accuracy of the estimates generated using this model. In general, to improve the accuracy of this approach, an application of similar distribution of operations to the ones we intend to evaluate needs to be selected. This is demonstrated in the experimental results in Chapter 4. When the average power of the Dhrystone benchmark [6] is used, this model estimates the energy of other benchmarks with intensive integer and memory operations like the Quicksort benchmarks with high accuracy. However, the accuracy of its estimates are greatly degraded when estimating the energy of different benchmarks the Quantization function.

2.1.3. First Order Instruction Level Model

In first order energy models, we assume constant energy required to execute each instruction. The model therefore contains an estimated energy of each instruction in a given processor instruction set. Using the parameters of a first order model and a list of instructions that execute in a given application, the energy of this application can be evaluated using equation (2.4). N denotes the total number of executed instructions of the application and $E(i)$ denotes the energy required to run the i^{th} instruction.

$$E_{application} = \sum_{i=1}^{N} E(i)\ 2 \quad (2.4)$$

When generating the first order model for the Microblaze processor, we followed the approach used in the literature, using infinite loops. First, we evaluated the energy of an empty infinite loop using a low-level energy estimation tool, namely the Xilinx XPA tool and post place and route models of the processor [4]. Then, we evaluate the energy of an infinite loop containing one instruction. Listing 2.1 shows the assembly instructions of the empty loop and the loop containing an *and* instructions. This method dictates we assume the difference between the energy of the two loops equal to the energy of the instruction. Applying this method to all Microblaze instructions, we obtain the parameters of the first order energy model as given in Table 2.1.

Listing 2.1 Assembly source code of empty infinite loop and infinite loop with an *and* instruction

Empty infinite loop

```
$L2:
     bri   $L2
```

and instruction in an infinite loop

```
$L2:
     and   r3,r4,r5
     bri   $L2
```

Table 2.1 Parameters of the Microblaze first order energy model

Instruction	Energy of each instruction (nJ)
add	1.1949
rsubk	1.3200
mul	1.2534
idiv	1.2934
and	1.3475
xori	1.1798
cmp	1.5821
nop	1.0698
lwi	1.4954
swi	1.4102
srl	1.2358
sra	1.2358

There is however a serious flaw in this method of approximating the energy of an instruction. The energy required to execute the empty is found to be only 0.76 nJ, which is very small in comparison to the energy required to execute the loop with an and instruction, estimated at 2.1 nJ. The reason for this large difference is clear when we examine the pipeline states as the Microblaze processor is executing each application. While executing a branch instruction, the Microblaze processor fetches two instructions before the branch instruction is decoded. When it is decoded, the pipeline is flushed, and the instruction at the target address is fetched. Figure 1.1 shows the Microblaze pipeline as it executes the empty loop given in Listing 2.1. In the first iteration, while the branch is fetched and decoded, the *rtsd* and *nop* instruction are fetched. However, when the branch is decoded and taken, they do not get into the execute stage. As a result, the branch instruction remains in the control and data-path of the last three pipeline stages. Therefore, the 0.76 nJ of

dynamic energy consumed by the processor is the result of the switching that occurs only in the first two stages. On the other hand, when the Microblaze is executing the infinite loop with the and instruction, the instruction in all pipeline stages change at least twice in each iterations as shown in Figure 2.2. Because of this phenomenon, the estimated energy of each instruction is greatly exaggerated. This is proven in the estimation results presented in chapter 4, that show an average error of 216% when estimates were obtained using these parameters. To work around this flaw, several papers suggest calibrating the estimates obtained using this model using the estimation error of one benchmark application [2] [24] [25] [27].

Loop Iteration	Instruction	Pipeline stages				
		IF	ID	EX	MEM	WB
1	**bri 0**	bri 0	X	X	X	X
	rtsd r17,0	rtsd r17,0	bri 0	X	X	X
	nop	nop	rtsd r17,0	bri 0	X	X
2	**bri 0**	bri 0	rtsd r17,0	bri 0	bri 0	X
	rtsd r17,0	rtsd r17,0	bri 0	bri 0	bri 0	bri 0
	nop	nop	rtsd r17,0	bri 0	bri 0	bri 0
3	**bri 0**	bri 0	rtsd r17,0	bri 0	bri 0	bri 0
	rtsd r17,0	rtsd r17,0	bri 0	bri 0	bri 0	bri 0
	nop	nop	rtsd r17,0	bri 0	bri 0	bri 0

Decoded branch causing pipeline flush | Stages not updated because of flushed instructions

Figure 2.1 Microblaze pipeline status during the execution of an empty inifinite loop

Loop Iteration	Instruction	Pipeline stages				
		IF	ID	EX	MEM	WB
1	**and r3,r4,r5**	and r3,r4,r5	X	X	X	X
	bri 0	bri 0	and r3,r4,r5	X	X	X
	rtsd r17,0	rtsd r17,0	bri 0	and r3,r4,r5	X	X
	and r3,r4,r5	and r3,r4,r5	rtsd r17,0	bri 0	and r3,r4,r5	X
2	**and r3,r4,r5**	and r3,r4,r5	rtsd r17,0	bri 0	bri 0	and r3,r4,r5
	bri 0	bri 0	and r3,r4,r5	bri 0	bri 0	bri 0
	rtsd r17,0	rtsd r17,0	bri 0	and r3,r4,r5	bri 0	bri 0
	and r3,r4,r5	and r3,r4,r5	rtsd r17,0	bri 0	and r3,r4,r5	bri 0

Decoded branch causing pipeline flush Stages not updated because of flushed instructions

Figure 2.2Microblaze pipeline status during the execution of an infinite loop with an and instruction

2.1.4. Second Order Instruction Level Model

Second order processor energy models increase the scope of the instruction-level energy model to incorporate inter-instruction energy effects. That is because, in addition to the type of instruction, the dynamic energy of an instruction also depends on the state of the processor prior to its execution. This inter-instruction effect can be accounted for in several ways. The most commonly used method, as described most related works [24] [25] [26] [27] [28], requires evaluating the energy of each pair of instructions. Knowing the energy of each possible pair of instructions, equation (2.5) can be used to estimate the energy required to execute a program**.** An $\boldsymbol{E(i,j)}$ term denotes the energy of the pair of instructions, i followed instruction j. Therefore $\boldsymbol{E(i+1,i)}$ denotes the energy required to execute a pair of consecutive instructions.

$$E_{application} = \frac{1}{2}\sum_{i=1}^{N-1} E(i+1,i) \qquad (2.5)$$

However, this approach requires evaluating a large number of instruction pairs using the method of infinite loops used to estimate the energy of instructions in the first order model. The same flaw discussed in the previous subsection applies to pairs of instructions in infinite loops. Furthermore, the most significant inter-instruction effect observed between pairs of instruction is a great reduction in the energy if a pair of the instruction that had the same opcode. Therefore, we decided to model the Microblaze processor using a simplified variation of second order models. Our model consists of energy estimates for instructions when they execute after instructions of the same type and a different estimate when they execute after instructions of the same type. The energy estimates as obtained for the first order model are taken as the energy estimates for instructions after different instruction types. Furthermore, we estimated the energy required to execute pairs of repeating instructions. We observed the dynamic energy required by the second instruction was consistently 55% of the energy required to execute a single instruction. This was consistent when the same instruction was repeated up to 7 times as can be seen in Figure 2.3. The figure shows the dynamic energy of load word, store word, shift right logic and shift right arithmetic instructions. These are the instructions that are used by the compiler in repeated sequences. Using this observation, we determined it is sufficient to implement the second order energy model using the parameters of the first order model and the energy drop ratio, as given in Table 2.2. Using these parameters, the energy required to execute a given application is evaluated using equation (2.6), where r is the ratio with which energy drops when the instruction is repeated.

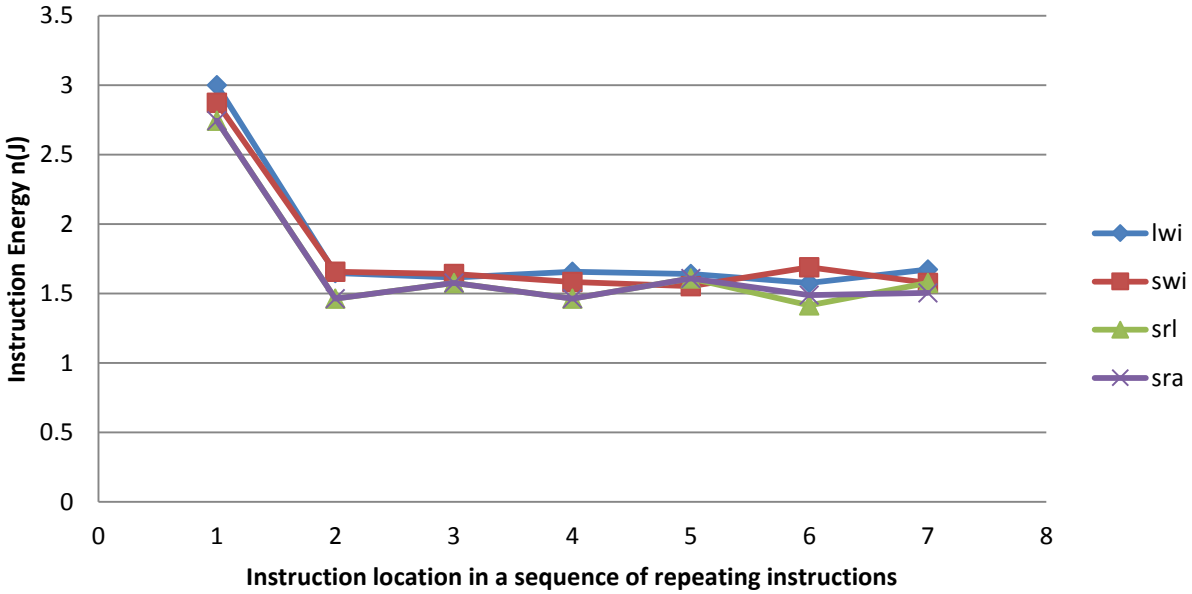


Figure 2.3 The dynamic energy of repeating instructions

Table 2.2 Parameters of the Microblaze second order energy model

Instruction	Energy of each instruction (nJ)
Repeated Instruction Energy Drop Ratio	0.5534
add	1.1949
rsubk	1.3200
mul	1.2534
idiv	1.2934
and	1.3475
xori	1.1798
cmp	1.5821
nop	1.0698
lwi	1.4954
swi	1.4102
srl	1.2358
sra	1.2358

$$E_{application} = \sum_{i=1}^{N} E(i) \cdot (inst_i \neq int_{i-1}) + E(i) \cdot (r) \cdot (inst_i == inst_{i-1}) \quad (2.6)$$

2.2 Operand Value Based Model

This section describes the methodology for creating the proposed Operand-Value-Based Model (OVBM) for a processor. The OVBM contents are as follows:

- A set of base energy costs for instructions,
- The maximum energy variance, due to operand values, for each instruction, and
- Two parameters, *m* and *b*, which are used to account for the effect of data values in the application.

The parameters of the OVBM, for a given soft-processor implementation, are extracted from the energy consumption of applications that are designed to reveal energy dissipation characteristics of different instructions, as well as the dependence of energy consumption on operand values. Post-place and route simulation tools and scripts are used to automate the generation of energy values. As an example, we have modeled a Microblaze soft processor implementation on a Xilinx Virtex 5 FPGA [6] using XPower Analyzer [4]. The Microblaze instruction set architecture is similar to the RISC-based DLX architecture [34].

2.2.1 Base Energy Cost of Instructions

We define the instruction base energy cost as the minimum dynamic energy needed for the processor to complete the instruction. It is determined by the type and operation of the instruction as well as the state of the processor's internal signals prior to executing the instruction. The state of the processor is determined by the preceding instruction(s), thus incorporating the inter-instruction energy effect. This can be done by simulating and tabulating the energy requirements for all possible combinations of instructions. However, the size of such a data model can get quite

large. To reduce the model size, we can group similar instructions in classes and model only the inter-class energy effects.

In order to characterize the energy requirements of instructions we have developed a novel technique. It is based on a fixed reference application. The purpose of this benchmark is to represent a sequence of diverse instructions that resembles a typical basic block, in which instructions are examined. This is to replace the use of infinite loops used to generate the parameters of the first and second order models, which are also commonly used in the related works. Listing 2.2 shows this benchmark containing the instructions and instruction constructs commonly found such as sequences of repeating shifts. The operands values of the instructions continuously change with each loop iteration. The energy of this loop is evaluated using low-level models.

Listing 2.2 Source code of the instruction energy characterization benchmark in C++ and compiler generated assembly

C++

```
int main()
{
int a=3,b=10,c=43;
     do{
          a=b*c;
          b=a^589994;
          c=a/b;
          b=c|19;
          c=a%16;
          c++;
     }while(c>1);
}
```

Compiler Generated Assembly

```
main:
...
      addik r3,r0,3     # 0x3
      swi   r3,r19,12
      addik r3,r0,10    # 0xa
      swi   r3,r19,8
      addik r3,r0,43    # 0x2b
      swi   r3,r19,4
$L2
      lwi   r4, r19, 8
      lwi   r3, r19, 4
      mul   r3, r4, r3
      swi   r3, r19, 12
      lwi   r3, r19, 12
      imm   9
      xor   r3, r3, 170
      swi   r3, r19, 8
      lwi   r4, r19, 12
      lwi   r3, r19, 8
      idiv  r3, r3, r4
      swi   r3, r19, 4
      lwi   r3, r19, 4
      ori   r3, r3, 19
      swi   r3, r19, 8
      lwi   r4, r19, 12
      addk  r3, r0, r4
      sra   r3, r4
      sra   r3, r3
      ...
      sra   r3, r3
      addk  r5, r0, r3
      srl   r5, r3
      srl   r5, r5
      ...
      srl   r5, r5
      addk  r3, r4, r5
      andi  r3, r3, 15
      rsubk r3, r5, r3
      swi   r3, r19, 4
      lwi   r3, r19, 4
      addik r3, r3, 1
      swi   r3, r19, 4
      lwi   r3, r19, 4
      addik r18, r0, 1
      cmp   r18, r3, r18
      blti  r18, $L2
```

Using the benchmarking application, a set of instruction benchmarking applications is created, one set of applications per instruction. In each application, an instruction with zero, or very small, operands is inserted between a pair of instructions in the reference application in Listing 2.2. Therefore, we create as many benchmarking applications per instruction as there are instructions in the reference application. In our modeling effort, we created a set of 37 benchmarking applications for each instruction to generate the Location Based Energy Profiles (LBEPs) for the Microblaze instruction set. The low-level model and estimation tool of the processor is then used to simulate the execution of the benchmarks and estimate the energy consumed. The differences between the energy consumed by the benchmarks with an added instruction and the reference application are evaluated. These differences approximate the energy required to execute the instruction when surrounded by different types of instructions. We refer to these differences as the *Location-Based Energy Profile* (LBEP) of the instruction. Instructions that utilize the same units of the processor data-path are expected to have similar LBEPs.

We derived LBEPs for the Microblaze instruction set, using accurate energy estimates from XPA [4]. They are given in Figure 2.4 through Figure 2.15. The X-axis in each LBEP shows, from left to right, the sequence of instructions in the reference application. The Y-axis is the increase in energy consumption when the instruction is inserted before the corresponding instruction on the X-axis. We identify three main types of instructions based on the data-path units they utilize. The groups are: arithmetic and logic, memory load and store, and shift. As expected, the instructions in each group have similar LBEPs, different from instruction in the other groups.

Figure 2.4 shows the LBEP of the addition (*add)* Microblaze instruction. It demonstrate the energy required to execute the set of benchmarking applications with added *(add)* instruction at different locations in the reference application. The instruction used is *addk r6, r7, r8*, where registers r7

and r8 are initialized to zero. The label ‘k’ in *(addk)* implies the instruction will keep the carry flag [35]. The LBEP shows the energy consumed by the *(addk)* instruction is lowest around other *(add)* instructions as well as at the beginning of the loop. It is also low around the *(mul), (imm), (idiv), (ori), (and), (rsub),* and *(cmp)* instructions; i.e. arithmetic and logic instructions. It is highest when the *(add)* instruction appears between two memory operations instructions; i.e. load word immediate *(lwi)* and store word immediate *(swi)* word instructions.

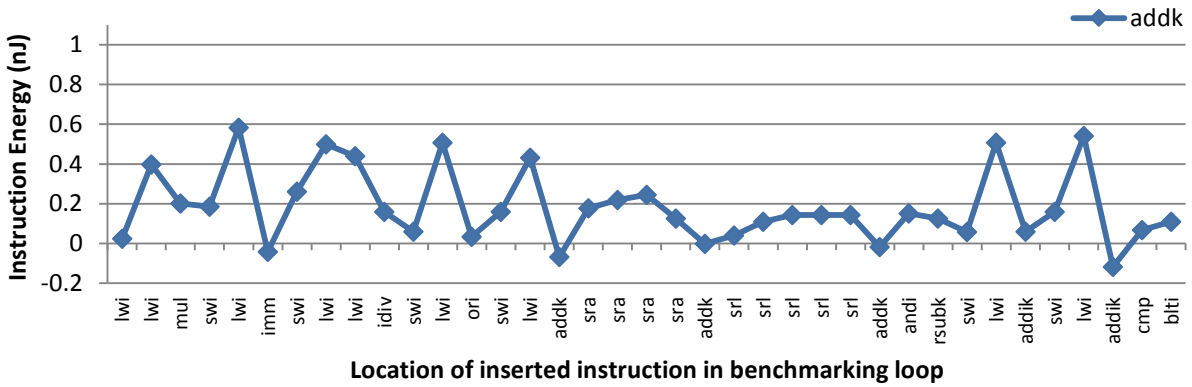


Figure 2.4 LBEP for the Microblaze *addk* instruction

Figure 2.5 shows the LBEP of the register subtraction *(rsub)* Microblaze instruction. It demonstrate the energy required to execute the set of benchmarking applications with added *(rsub)* instruction at different locations in the reference application. The instruction used is *rsubk r8, r6, r7*, where registers r6 and r7 are initialized to zero. The label ‘k’ in *(rsubk)* implies the instruction will keep the carry flag [35]. The LBEP shows the energy consumed by the *(rsubk)* instruction is minimum around the *(mul), (imm), (idiv), (ori), (add), (and),* and *(cmp)* instructions; i.e. arithmetic and logic instructions. It is highest when the (rsub) instruction appears between two memory operation or shift instructions; i.e. load *(lwi)* and store *(swi)* word instructions, and shift right logic *(srl)* and arithmetic *(sra)* instructions.

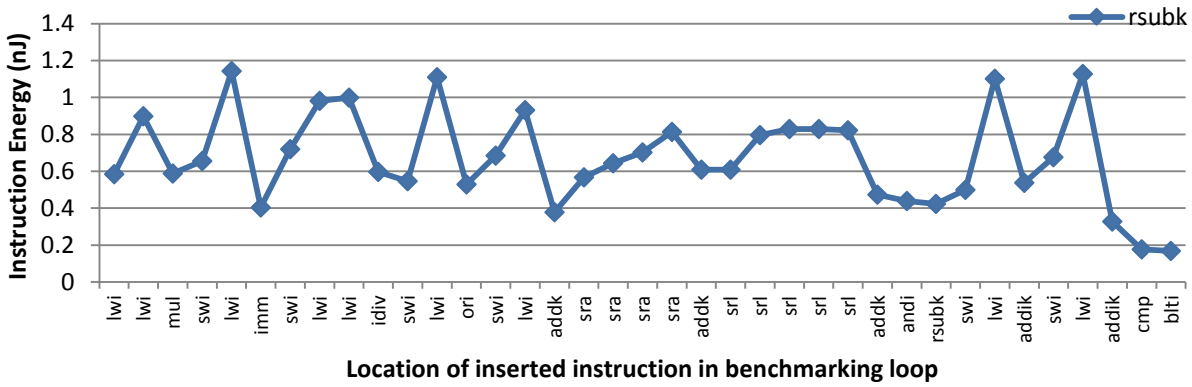


Figure 2.5 LBEP for the Microblaze *rsubk* instruction

Figure 2.6 shows the LBEP of the integer multiplication (*muli)* Microblaze instruction. It demonstrate the energy required to execute the set of benchmarking applications with added *(muli)* instruction at different locations in the reference application. The instruction used is *muli r6, r6, 5*, where register r6 is initialized to zero. The LBEP shows the energy consumed by the *(muli)* instruction is minimum around low around the *(imm), (idiv), (ori), (add), (and), (rsub),* and *(cmp)* instructions; i.e. arithmetic and logic instructions. It is highest when the *(muli)* instruction appears between two memory operation or shift instructions; i.e. load *(lwi)* and store *(swi)* word instructions, and shift right logic *(srl)* and arithmetic *(sra)* instructions.

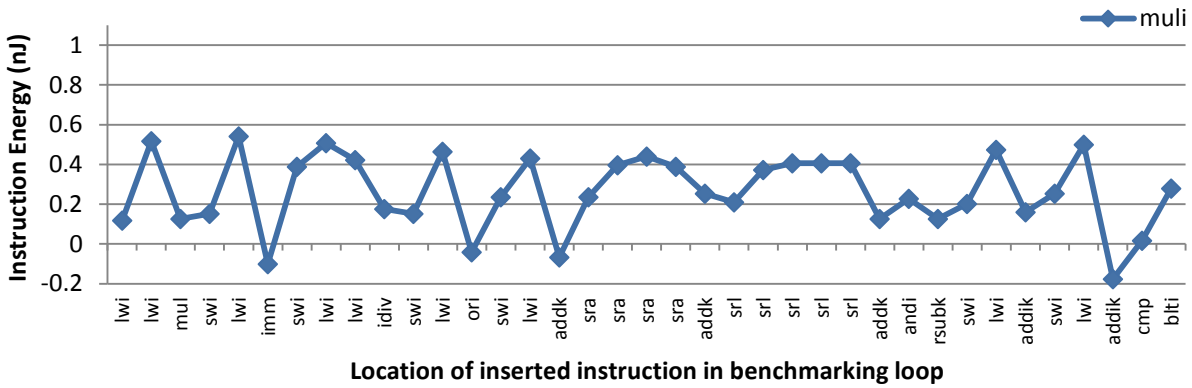


Figure 2.6 LBEP for the Microblaze *muli* instruction

Figure 2.7 shows the LBEP of the (idiv) Microblaze instruction. It demonstrate the energy required to execute the set of benchmarking applications with added *(idiv)* instruction at different locations in the reference application. The instruction used is *idiv r7, r8, r6*, where registers r8 and r6 are initialized to 5 and 19 respectively. The instruction therefore repeatedly performs an integer division of 19 over 5 [35]. The LBEP shows the energy consumed by the *(idiv)* instruction is lowest around other *(idiv)* instructions as well as at the beginning of the loop and before the immediate instruction. It is also significantly low around the *(mul), (imm), (ori), (add), (and), (rsub),* and *(cmp)* instructions; i.e. arithmetic and logic instructions. It is highest when the (idiv) instruction appears between two memory operation or shift instructions; i.e. load *(lwi)* and store *(swi)* word instructions, and shift right logic *(srl)* and arithmetic *(sra)* instructions.

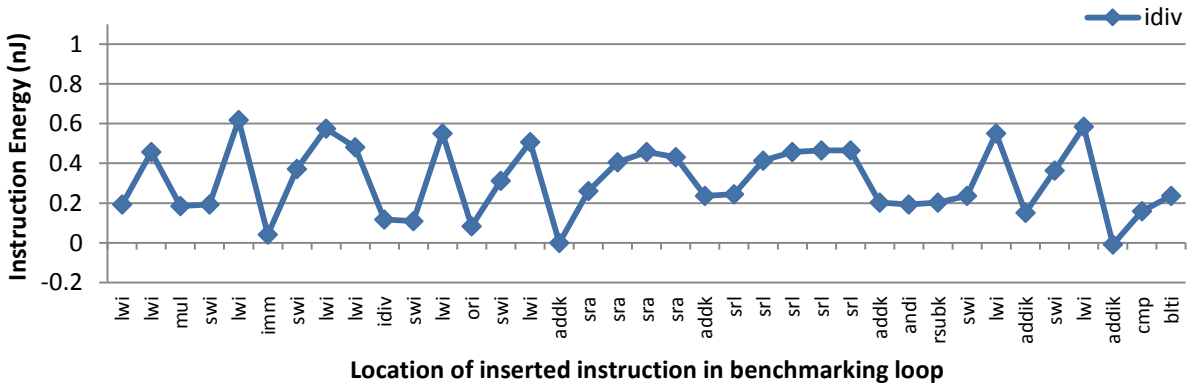


Figure 2.7 LBEP for the Microblaze *idiv* instruction

Figure 2.8 shows the LBEP of the logic-and *(and)* Microblaze instruction. It demonstrate the energy required to execute the set of benchmarking applications with added *(and)* instruction at different locations in the reference application. The instruction used is *andi r6, r6, 13*, where register r6 is initialized to 19. The LBEP shows the energy consumed by the *(andi)* instruction is lowest around the *(mul), (imm), (idiv), (ori), (add), (rsub),* and *(cmp)* instructions; i.e. arithmetic and logic instructions. It is highest when the (and) instruction appears between two memory

operation or shift instructions; i.e. load *(lwi)* and store *(swi)* word instructions, and shift right logic *(srl)* and arithmetic *(sra)* instructions.

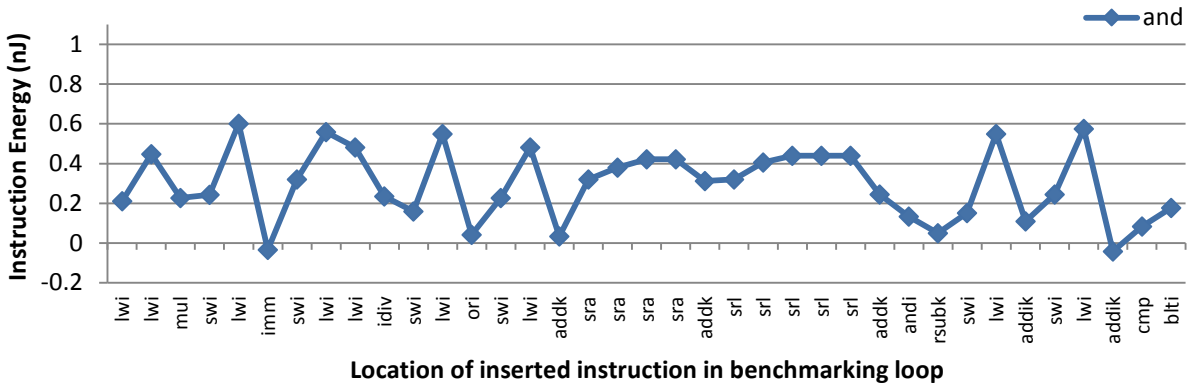


Figure 2.8 LBEP for the Microblaze *and* instruction

Figure 2.9 shows the LBEP of the logic exclusive or *(xor)* Microblaze instruction. It demonstrate the energy required to execute the set of benchmarking applications with added *(xor)* instruction at different locations in the reference application. The instruction used is *xori r7, r6, 207*, where register r6 is initialized to 19. The LBEP shows the energy consumed by the *(xor)* instruction is lowest around the *(mul), (imm), (idiv), (ori), (add), (and), (rsub),* and *(cmp)* instructions; i.e. arithmetic and logic instructions. It is highest when the *(xor)* instruction appears between two memory operation or shift instructions; i.e. load *(lwi)* and store *(swi)* word instructions, and shift right logic *(srl)* and arithmetic *(sra)* instructions.

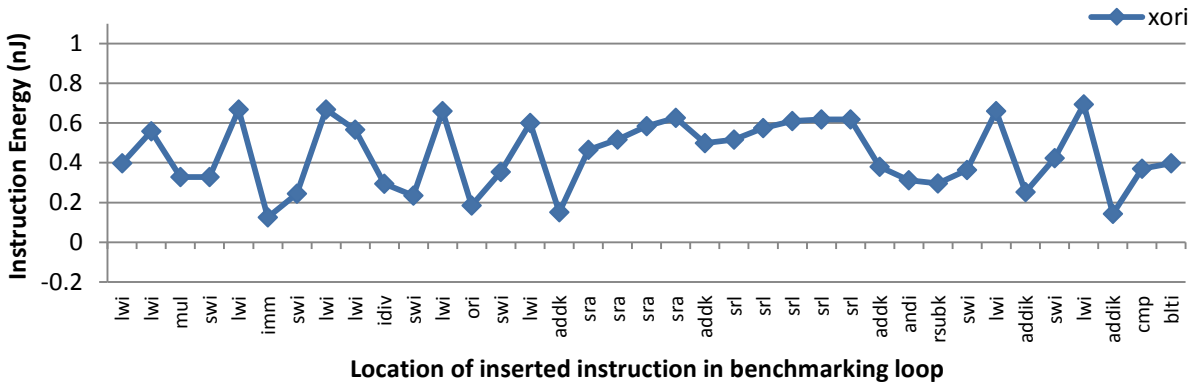


Figure 2.9 LBEP for the Microblaze *xori* instruction

Figure 2.10 shows the LBEP of the compare *(cmp)* Microblaze instruction. It demonstrate the energy required to execute the set of benchmarking applications with added *(cmp)* instruction at different locations in the reference application. The instruction used is *cmp r8, r7, r6*, where registers r6 and r7 are initialized to 19 and 13 respectively. The instruction hence subtracts 13 from 19 resulting in 6, which only contains 2 ones [35]. The LBEP shows the energy consumed by the *(cmp)* instruction is lowest around other *(cmp)* instructions as well as at the beginning of the loop and before the immediate instruction. It is also significantly low around the *(mul), (imm), (idiv), (ori), (add), (and),* and *(rsub)* instructions; i.e. arithmetic and logic instructions. It is highest when the *(cmp)* instruction appears between two memory operation or shift instructions; i.e. load *(lwi)* and store *(swi)* word instructions, and shift right logic *(srl)* and arithmetic *(sra)* instructions.

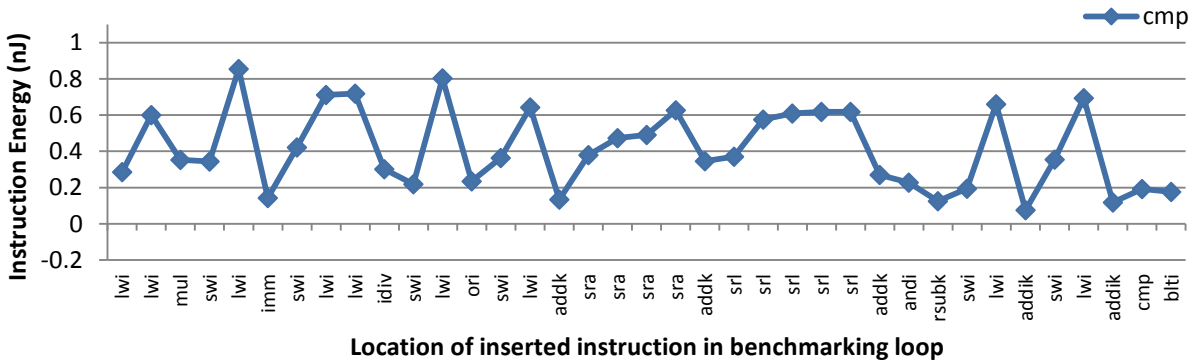


Figure 2.10 LBEP for the Microblaze *cmp* instruction

Figure 2.11 shows the LBEP of the no-operation (*nop)* Microblaze instruction. It demonstrate the energy required to execute the set of benchmarking applications with added *(nop)* instruction at different locations in the reference application. It is important to note the *(nop)* instruction is implemented using an *(or)* instruction, namely as *or r0, r0, r0* [35]. Therefore, the *(nop)* is considered a logic operation instruction. The LBEP shows the energy consumed by the *(nop)* instruction is lowest around the *(mul), (imm), (idiv), (ori), (add), (and), (rsub),* and *(cmp)* instructions; i.e. arithmetic and logic instructions. It is highest when the *(nop)* instruction appears between two memory operations instructions; i.e. load *(lwi)* and store *(swi)* word instructions.

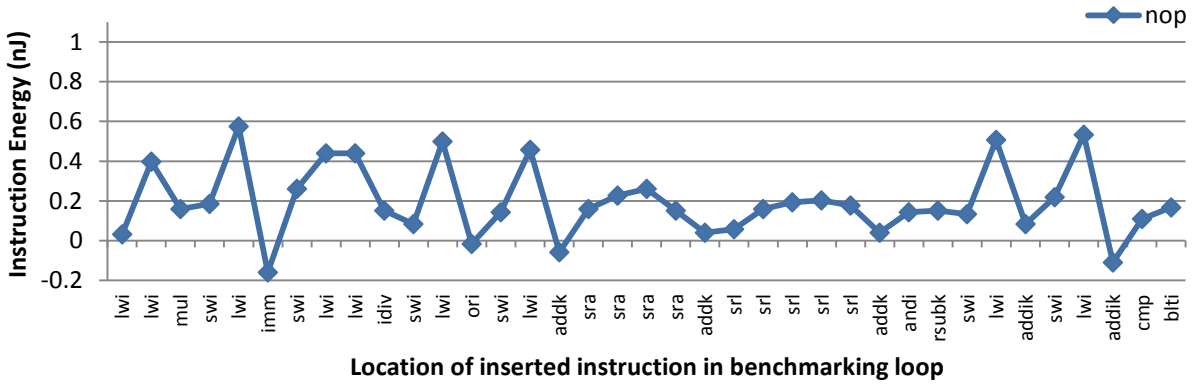


Figure 2.11 LBEP for the Microblaze *nop* instruction

Figure 2.12 shows the LBEP of the load word from memory (lwi) Microblaze instruction. It demonstrate the energy required to execute the set of benchmarking applications with added *(lwi)* instruction at different locations in the reference application. The instruction used is *lwi r6, r19, 16*, where the value of r9 plus 16 is the memory location of a variable initialized to zero. The instruction therefore loads a zero value into register r6. The LBEP shows the energy consumed by the *(lwi)* instruction is lowest around other *(lwi)* and (swi) instructions; i.e. other memory instructions. It is highest when the (lwi) instruction appears between shift instruction and logic and arithmetic instructions.

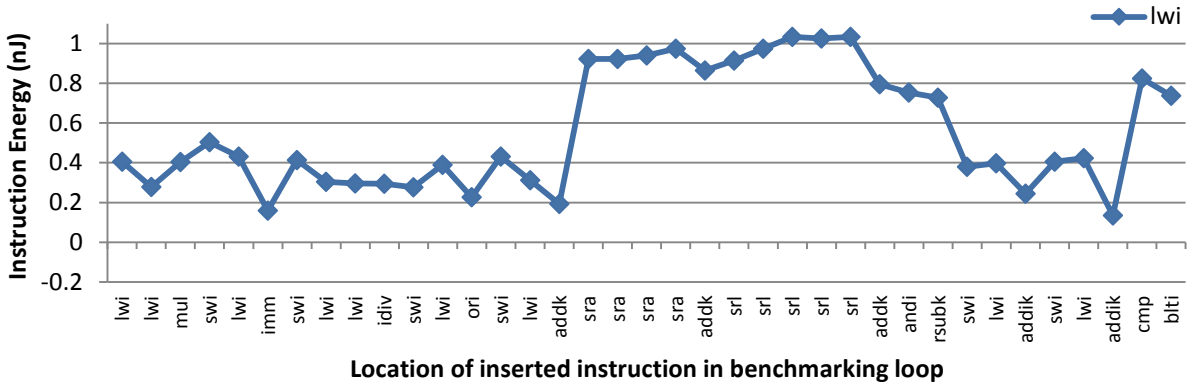


Figure 2.12 LBEP for the Microblaze *lwi* instruction

Figure 2.13 shows the LBEP of the store word into memory (swi) Microblaze instruction. It demonstrate the energy required to execute the set of benchmarking applications with added *(swi)* instruction at different locations in the reference application. The instruction used is *swi r0, r19, 4*, which stores the zero value of r0 in the memory location of the value of r19 plus 4. The LBEP shows the energy consumed by the *(swi)* instruction is lowest around other *(swi) and (lwi)* instructions; i.e. other memory instructions. It is highest when the (swi) instruction appears between shift instruction and logic and arithmetic instructions.

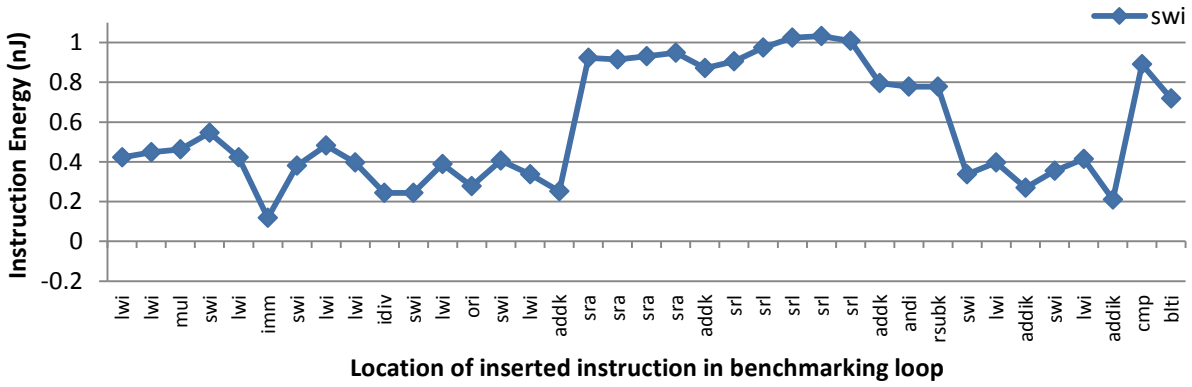


Figure 2.13 LBEP for the Microblaze *swi* instruction

Figure 2.14 shows the LBEP of the shift right logic (srl) Microblaze instruction. It demonstrate the energy required to execute the set of benchmarking applications with added *(srl)* instruction at

different locations in the reference application. The instruction used is *srl r6, r8,* where registers r6 and r8 are initialized to zero. The LBEP shows the energy consumed by the *(srl)* instruction is lowest around other shift instructions. It is also significantly low around the *(imm),* and *(add)* instructions. It is highest when the (srl) instruction appears between two memory operations instructions; i.e. load *(lwi)* and store *(swi)* word instructions. The energy of the shift instruction is at an intermediate level when the instruction appears around *(mul), (idiv), (and),* and *(rsub)* instructions.

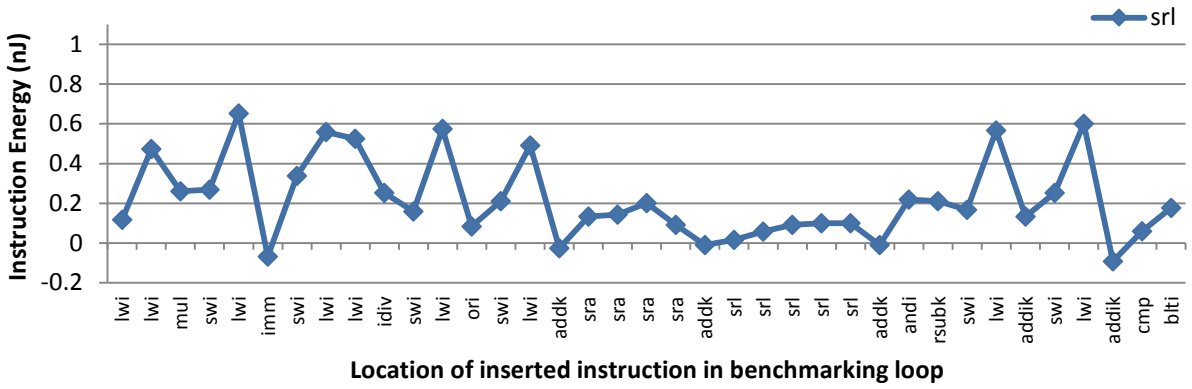


Figure 2.14 LBEP for the Microblaze *srl* instruction

Figure 2.15 shows the LBEP of the shift right arithmetic (sra) Microblaze instruction. It demonstrate the energy required to execute the set of benchmarking applications with added *(sra)* instruction at different locations in the reference application. The instruction used is *sra r6, r8*, where registers r6 and r8 are initialized to zero. The LBEP shows the energy consumed by the *(sra)* instruction is lowest other shift instructions. It is also significantly low around the *(imm),* and *(add)* instructions. It is highest when the (sra) instruction appears between two memory operations instructions; i.e. load *(lwi)* and store *(swi)* word instructions. The energy of the shift instruction is at an intermediate level when the instruction appears around *(mul), (idiv), (and),* and *(rsub)* instructions.

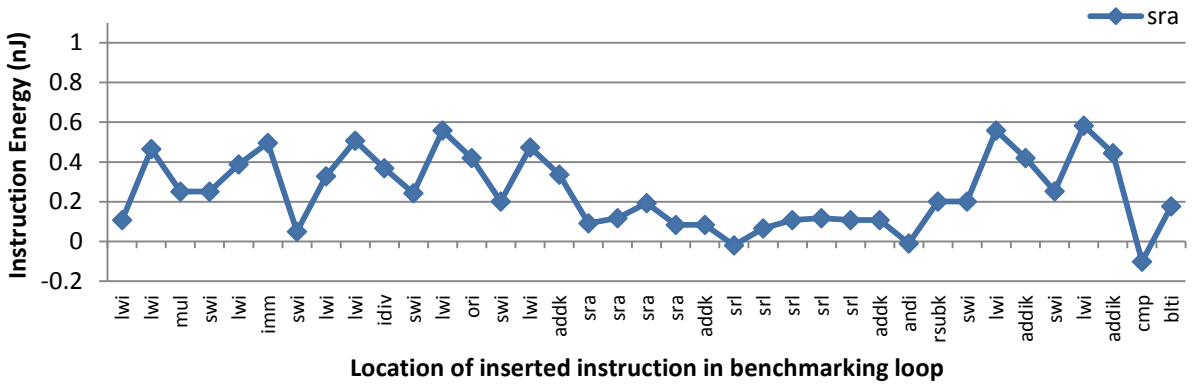


Figure 2.15 LBEP for the Microblaze *sra* instruction

The LBEPs help identify the classes of instructions but their values do not directly represent the energy cost of the inserted instruction. Consider the portion of the reference loop with and without an inserted "addk" instruction as shown in Figure 2.16. The energy consumed by executing the code on the RHS includes the energy to complete the (*addk*) instruction after the store instruction (*swi*). It also includes the energy consumed to decode the load instruction (*lwi*) after the inserted (*addk*). However, the energy consumed to execute the reference application on the LHS includes the energy to decode a load instruction after a store instruction. Therefore, the LBEP value obtained by finding the difference between the energy consumption of LHS and RHS code segments can be expressed in (2.5) denoted by $\Delta E(addk)$.

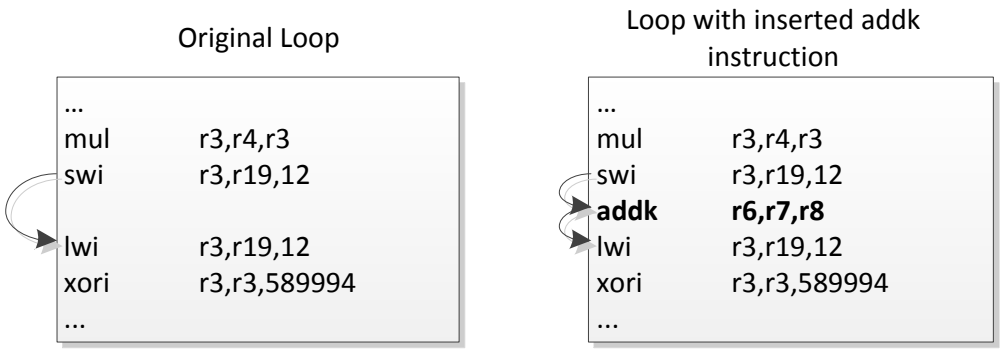


Figure 2.16 Reference Loop With/Without inserted *addk* instruction

$$\Delta E(\boldsymbol{addk}) = \boldsymbol{E}\,(\boldsymbol{addk\ after\ store}) - \boldsymbol{E_{decode}}\,(\boldsymbol{load\ after\ store}) \qquad \textbf{(2.7)}$$

It is difficult to evaluate the energy required to decode each instruction of the reference benchmarking application, taking into account its preceding instruction. However, the LBEPs presented using low-level power simulations indicate that E_{decode} for an instruction is very small when it follows an instruction from the same group. On the other hand, it might be larger than the energy of the inserted instruction. In such cases, the energy difference would produce a negative energy value on the LBEP as can be seen in some of the preceding LBEP figures. Therefore, in order to calculate the base energy cost of an instruction, we consider only those sample points in the LBEP in which the instruction is inserted between two instructions of the same type. Hence, we obtain three base energy costs for each instruction, one for each case where it executes following instructions of a specific group. For instance, from the energy profile of the load word instruction (*lwi*) given in Figure 1, we first consider the energy values corresponding to the (*lwi*) instruction inserted between pairs of logic or arithmetic instructions. The average of these energy estimates is recorded as the base energy cost of the load instruction following a logic or arithmetic operation. Similarly, estimates for the base energy cost of the lwi instruction following memory and shift instructions, are also evaluated. The three base energy costs of select Microblaze

instructions are presented in Table 2.3 for when they follow an instruction from one of the three groups. This constitutes the first component of the OVBM.

Table 2.3 Base energies variations for Microblaze instructions in Nano Joules

Instruction	Base Energy After instruction from class (nJ)		
	Arithmetic & Logic	***Memory***	***Shift***
add	0.1147	0.4882	0.1608
rsubk	0.3461	1.0352	0.7762
mul	0.1233	0.4819	0.4019
idiv	0.1850	0.5401	0.4419
and	0.0892	0.5306	0.4213
xori	0.3257	0.6345	0.5921
cmp	0.1821	0.7108	0.5727
nop	0.1343	0.4808	0.1959
lwi	0.7680	0.3536	0.9858
swi	0.8159	0.4108	0.9761
srl	0.1628	0.5550	0.1124
sra	0.1571	0.5836	0.1899

2.2.2 Maximum Energy Variance

The dynamic power consumed by the soft processor cores implemented on an FPGA depends on the operand values of the instruction. This is because internal signals in processor cores implemented in FPGAs are often much longer than those in ASIC implementations. The operand values of instructions need to propagate between the data-path units implemented on several CLBs in FPGA implementations. On the other hand, data-path units in ASIC processors are placed within close proximity of one another to minimize latency and power consumption. Although dynamic energy is consumed by the signals switching, the number of ones in the values of the operands is

a good indication of the possibility of switching. Because instructions such as NOPs and branches have zero operands and are executed frequently, the more ones in the operands of an instruction, the more likely it is to cause higher dynamic energy consumption. This is demonstrated in the results presented in this section.

To model the influence of operand values, we introduce a new parameter to the OVBM, called maximum instruction energy variance. It is defined as the maximum difference between the energy cost of an instruction with large operands and the instruction's base energy cost. To observe this variance, a copy of the energy benchmarks used to generate the base LBEPs, described in the previous subsection, is created. The operand values of the inserted instructions are set to the maximum positive values of 0x7fffffff instead of zero. The process to generate the LBEPs is repeated, generating a new maximum LBEP for each instruction.

Figure 2.17 shows the minimum and maximum LBEP for the *(add)* Microblaze instruction. The lower graph, the minimum LBEP, was used to calculate the base energy of the *(add)* instruction. The upper graph represents the energy required to execute the instruction with maximum operand values at the different locations in the reference benchmarking application. The difference between the maximum and minimum LBEPs is presented by the bar graph. The average of these differences is taken as the maximum, operand value dependent, energy variance of the *(add)* instruction.

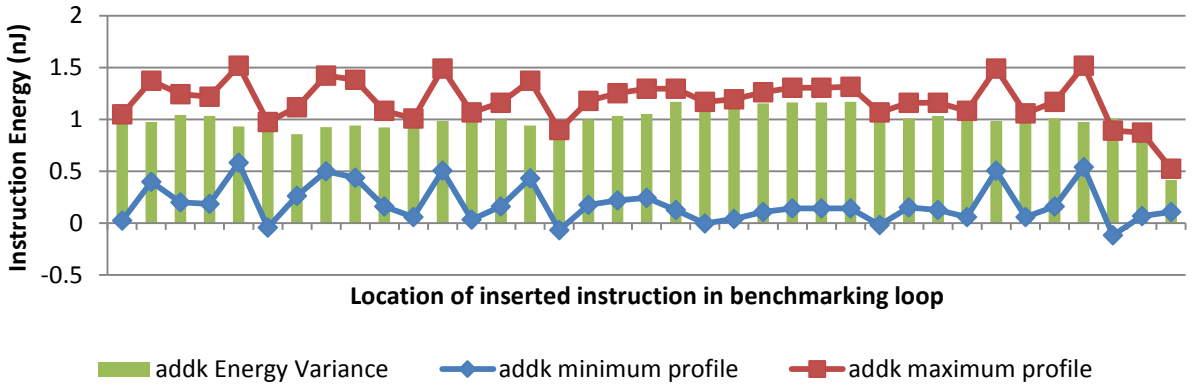


Figure 2.17 Maximum and minimum energy profiles and maximum energy variance for the *addk* instruction

Figure 2.18 shows the minimum and maximum LBEP for the *(rsub)* Microblaze instruction. The lower graph, the minimum LBEP, was used to calculate the base energy of the *(rsub)* instruction. The upper graph represents the energy required to execute the instruction with maximum operand values at the different locations in the reference benchmarking application. The difference between the maximum and minimum LBEPs is presented by the bar graph. The average of these differences is taken as the maximum, operand value dependent, energy variance of the *(rsub)* instruction.

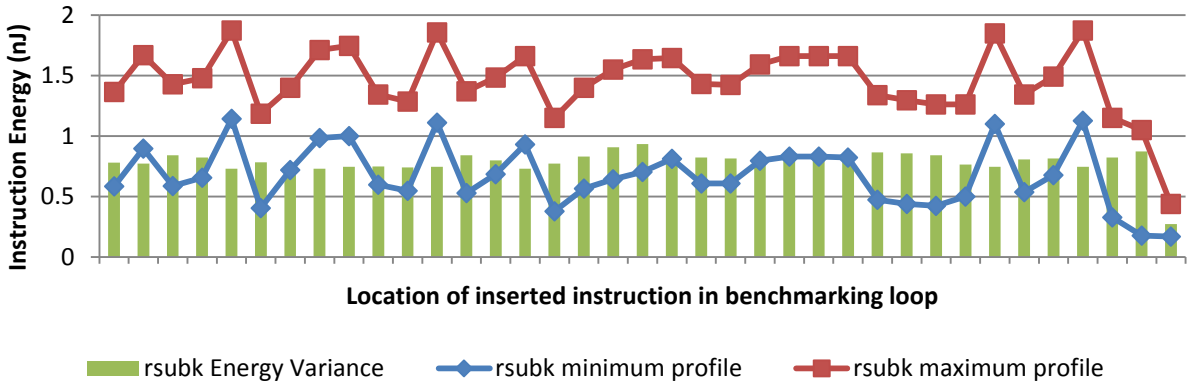


Figure 2.18 Maximum and minimum energy profiles and maximum energy variance for the rsubk instruction

Figure 2.19 shows the minimum and maximum LBEP for the *(mul)* Microblaze instruction. The lower graph, the minimum LBEP, was used to calculate the base energy of the *(mul)* instruction. The upper graph represents the energy required to execute the instruction with maximum operand values at the different locations in the reference benchmarking application. The difference between the maximum and minimum LBEPs is presented by the bar graph. The average of these differences is taken as the maximum, operand value dependent, energy variance of the *(mul)* instruction.

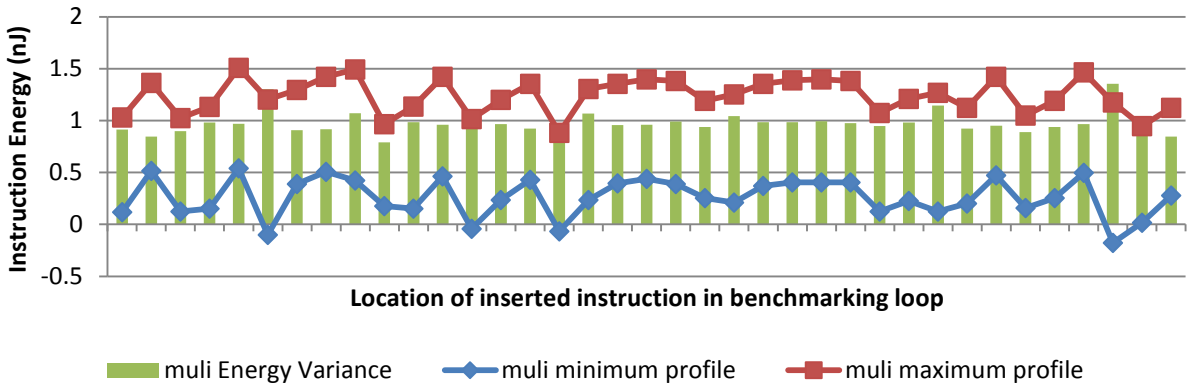


Figure 2.19 Maximum and minimum energy profiles and maximum energy variance for the *muli* instruction

Figure 2.20 shows the minimum and maximum LBEP for the *(idiv)* Microblaze instruction. The lower graph, the minimum LBEP, was used to calculate the base energy of the *(idiv)* instruction. The upper graph represents the energy required to execute the instruction with maximum operand values at the different locations in the reference benchmarking application. The difference between the maximum and minimum LBEPs is presented by the bar graph. The average of these differences is taken as the maximum, operand value dependent, energy variance of the *(idiv)* instruction.

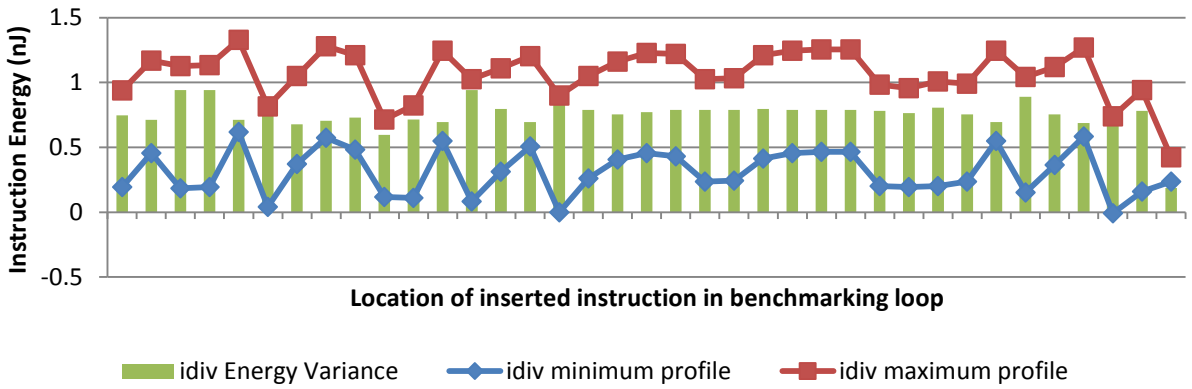


Figure 2.20 Maximum and minimum energy profiles and maximum energy variance for the *idiv* instruction

Figure 2.21 shows the minimum and maximum LBEP for the *(and)* Microblaze instruction. The lower graph, the minimum LBEP, was used to calculate the base energy of the *(and)* instruction. The upper graph represents the energy required to execute the instruction with maximum operand values at the different locations in the reference benchmarking application. The difference between the maximum and minimum LBEPs is presented by the bar graph. The average of these differences is taken as the maximum, operand value dependent, energy variance of the *(and)* instruction.

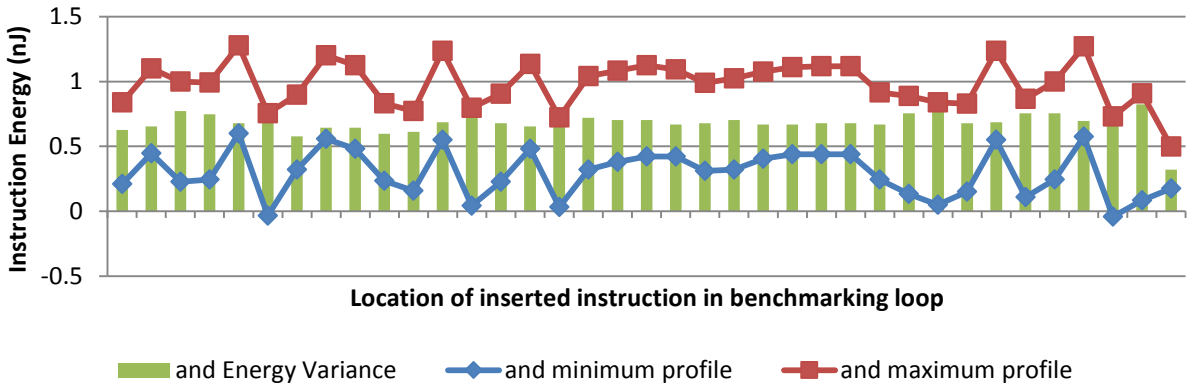


Figure 2.21 Maximum and minimum energy profiles and maximum energy variance for the *and* instruction

Figure 2.22 shows the minimum and maximum LBEP for the *(xor)* Microblaze instruction. The lower graph, the minimum LBEP, was used to calculate the base energy of the *(xor)* instruction. The upper graph represents the energy required to execute the instruction with maximum operand values at the different locations in the reference benchmarking application. The difference between the maximum and minimum LBEPs is presented by the bar graph. The average of these differences is taken as the maximum, operand value dependent, energy variance of the *(xor)* instruction.

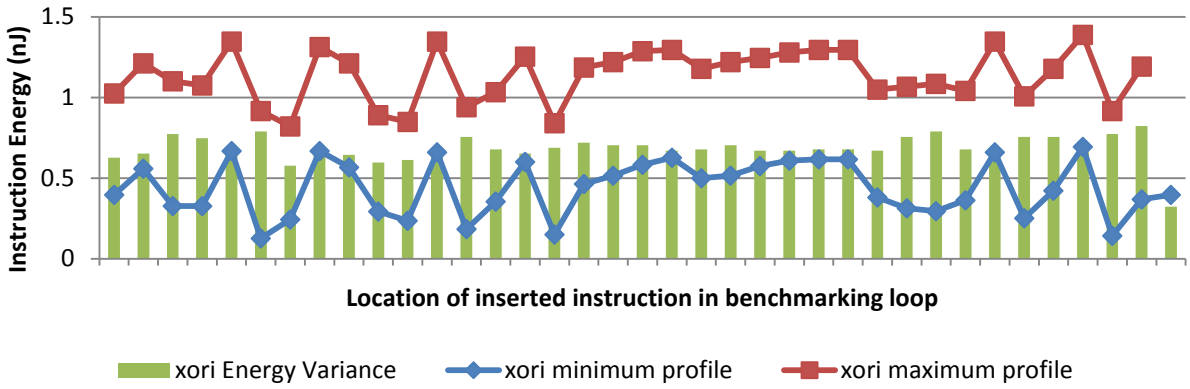


Figure 2.22 Maximum and minimum energy profiles and maximum energy variance for the *xori* instruction

Figure 2.23 shows the minimum and maximum LBEP for the *(cmp)* Microblaze instruction. The lower graph, the minimum LBEP, was used to calculate the base energy of the *(cmp)* instruction. The upper graph represents the energy required to execute the instruction with maximum operand values at the different locations in the reference benchmarking application. The difference between the maximum and minimum LBEPs is presented by the bar graph. The average of these differences is taken as the maximum, operand value dependent, energy variance of the *(cmp)* instruction.

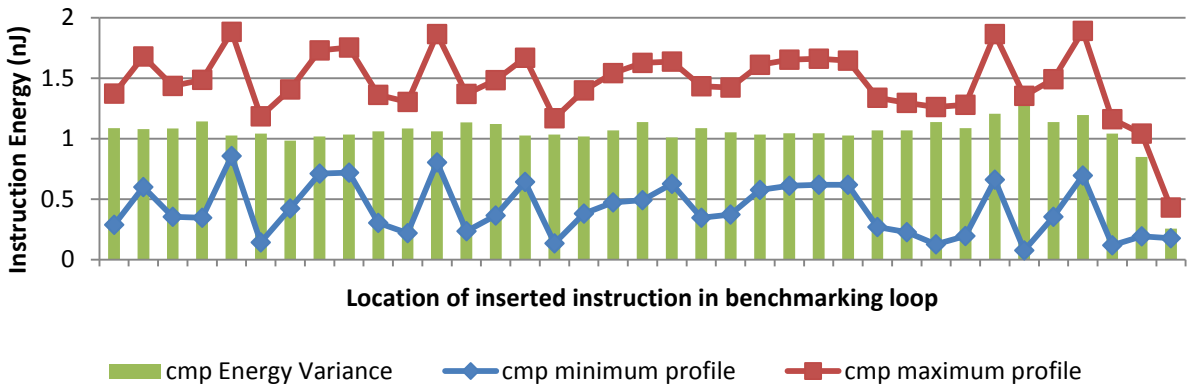


Figure 2.23 Maximum and minimum energy profiles and maximum energy variance for the *cmp* instruction

Figure 2.24 shows the minimum and maximum LBEP for the *(lwi)* Microblaze instruction. The lower graph, the minimum LBEP, was used to calculate the base energy of the *(lwi)* instruction. The upper graph represents the energy required to execute the instruction with maximum operand values at the different locations in the reference benchmarking application. The difference between the maximum and minimum LBEPs is presented by the bar graph. The average of these differences is taken as the maximum, operand value dependent, energy variance of the *(lwi)* instruction.

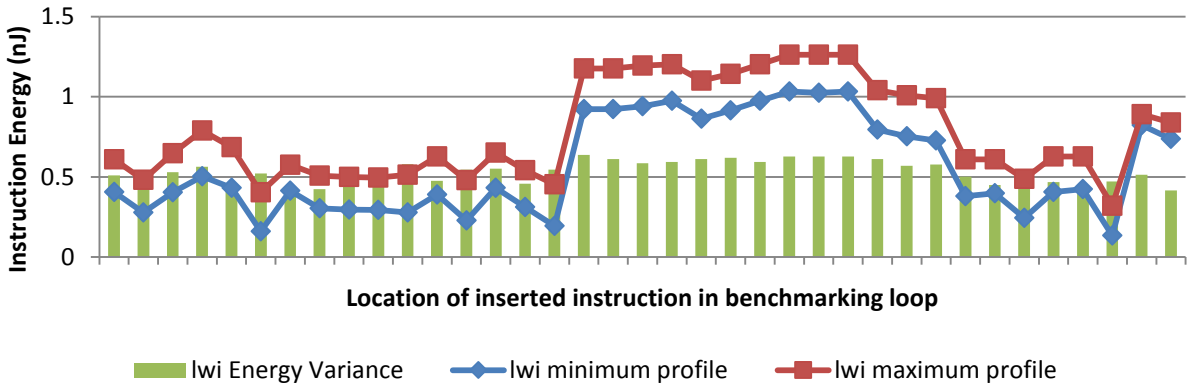


Figure 2.24 Maximum and minimum energy profiles and maximum energy variance for the *lwi* instruction

Figure 2.25 shows the minimum and maximum LBEP for the *(swi)* Microblaze instruction. The lower graph, the minimum LBEP, was used to calculate the base energy of the *(swi)* instruction. The upper graph represents the energy required to execute the instruction with maximum operand values at the different locations in the reference benchmarking application. The difference between the maximum and minimum LBEPs is presented by the bar graph. The average of these differences is taken as the maximum, operand value dependent, energy variance of the *(swi)* instruction.

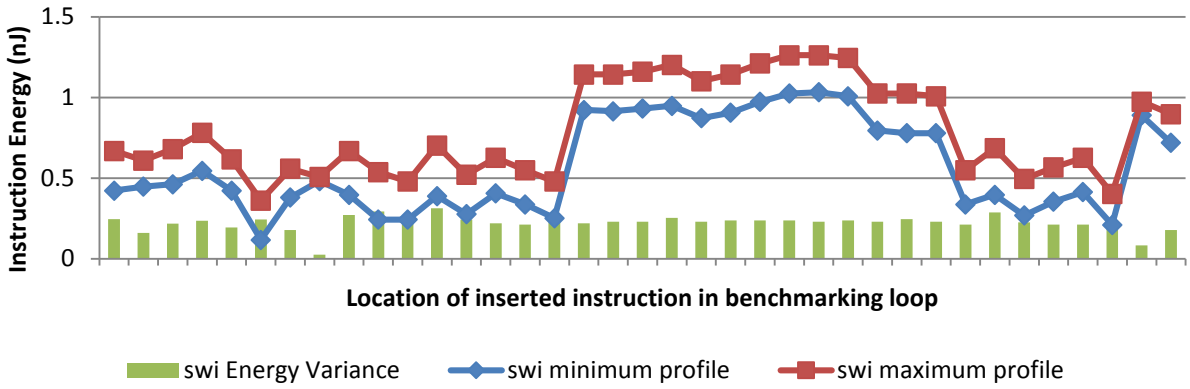


Figure 2.25 Maximum and minimum energy profiles and maximum energy variance for the *swi* instruction

Figure 2.26 shows the minimum and maximum LBEP for the *(srl)* Microblaze instruction. The lower graph, the minimum LBEP, was used to calculate the base energy of the *(srl)* instruction. The upper graph represents the energy required to execute the instruction with maximum operand values at the different locations in the reference benchmarking application. The difference between the maximum and minimum LBEPs is presented by the bar graph. The average of these differences is taken as the maximum, operand value dependent, energy variance of the *(srl)* instruction.

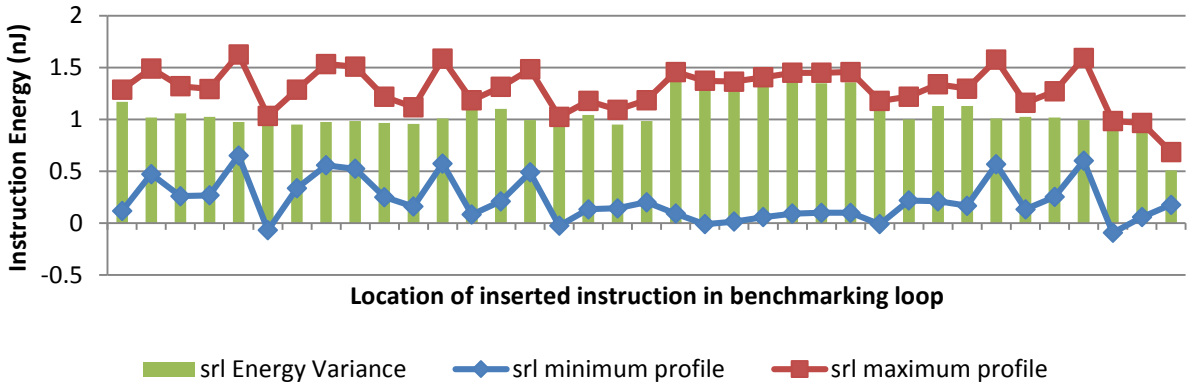


Figure 2.26 Maximum and minimum energy profiles and maximum energy variance for the *srl* instruction

Figure 2.27 shows the minimum and maximum LBEP for the *(sra)* Microblaze instruction. The lower graph, the minimum LBEP, was used to calculate the base energy of the *(sra)* instruction. The upper graph represents the energy required to execute the instruction with maximum operand values at the different locations in the reference benchmarking application. The difference between the maximum and minimum LBEPs is presented by the bar graph. The average of these differences is taken as the maximum, operand value dependent, energy variance of the *(sra)* instruction.

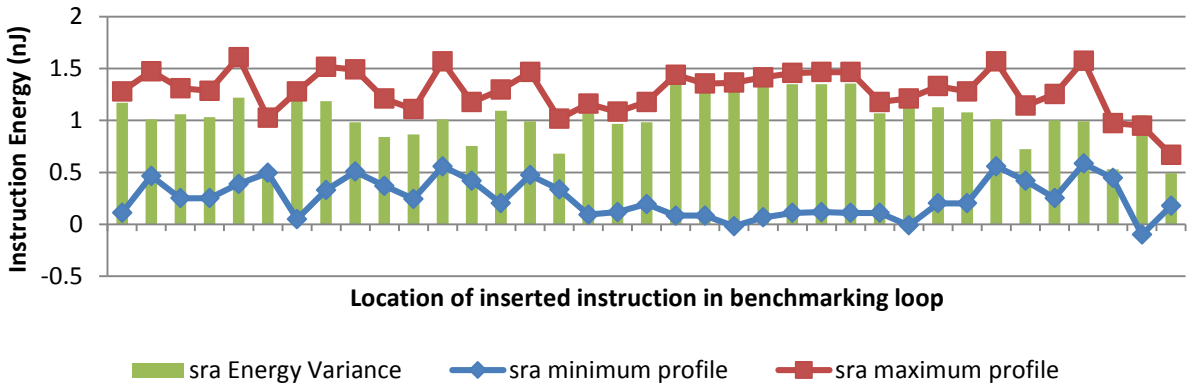


Figure 2.27 Maximum and minimum energy profiles and maximum energy variance for the *sra* instruction

As seen in figures, the difference between the maximum energy and the base energy does not vary significantly with the location of the inserted instruction. This is consistent with most Microblaze instructions. The small variance is due to the different average one's densities of the instructions of the reference loop. This variation of operand values is also difficult to estimate in applications as their energy cost is being estimated. Therefore, the differences between each set of two LBEPs are averaged to obtain a single value of maximum energy variance for each instruction. These values are appended to the base energy estimates as given in Table 2.4.

The total energy consumed by an instruction i is therefore modeled using equation (2.6). $E_{base}(i,j)$ is the base energy of instruction i following instruction j. $EV(i)$ is the maximum energy variance of instruction i. Finally, k is a factor that determines the specific energy variance of instruction i, to be derived from its operands value. This factor will be discussed in the following subsection.

$$\boldsymbol{E_i = E_{base}(i,j) + k \cdot EV(i)} \qquad \textbf{(2.8)}$$

Table 2.4 Base energies and maximum energy variations for Microblaze instructions in Nano Joules

Instruction	Base energy after instruction from class (nJ)			Max. instr. Energy Variance
	Arithmetic & Logic	*Memory*	*Shift*	
add	0.1147	0.4882	0.1608	1.0034
rsubk	0.3461	1.0352	0.7762	0.7872
mul	0.1233	0.4819	0.4019	0.9795
idiv	0.1850	0.5401	0.4419	0.7602
and	0.0892	0.5306	0.4213	0.6977
xori	0.3257	0.6345	0.5921	0.6977
cmp	0.1821	0.7108	0.5727	1.0456
nop	0.1343	0.4808	0.1959	0
lwi	0.7680	0.3536	0.9858	0.5310
swi	0.8159	0.4108	0.9761	0.2208
srl	0.1628	0.5550	0.1124	1.0782
sra	0.1571	0.5836	0.1899	1.0373

2.2.3 Energy impact of operand values

As can be seen in the results presented in the previous section, the operand value have a significant effect of the energy consumed by soft-processors in FPGA. The following subsections define the impact of the operand values on the energy consumption of two types of instruction structures. In heterogeneous sequences of instructions, the more ones in the instruction operands, the more likely it is that internal signals will be switched. However, in homogeneous sequences of repeating shift instructions, the energy impact of operand values depends on the number of alternating bit values.

2.2.3.1 Energy Impact of Operand Density

To confirm the hypothesis that one's density in operand values of instructions in heterogeneous sequences impacts the energy consumed, we used a new set of benchmarks. An application containing a source array of 10 elements and a loop with a fixed set of different and non-repeating instructions is created as given in Listing 2.3. The instructions in the loop load a value from the source array and perform several operations using it. Copies of this application are generated and refactored by changing the values in the source array. In each application, the array is initialized to integers containing the same number of ones, different from the other applications. For instance, the first application operates on source array with values that are all powers of two as shown in Listing 2.3. In the second application, the source array is initialized to the 10 values that include 33554433, 67109888, and 524416, all 32-bit positive integers containing exactly two bits with the logic 1 value. In total 30 applications are created, and the energy required to execute each is estimated using post-place and route tools.

Listing 2.3 First application used to observe dependency of energy consumed on one's densities

```
#define size 10
int main(){
     int temp, arr_in[size]=
{1024, 4194304, 67108864, 2048, 128, 256, 2};
     while(1){
          for (int i=0; i<size; i++){
               temp=arr_in[i];
               temp*=2;
               temp++;
          }
     }
     return 0;
}
```

To model the dependence of the energy consumption on operand density for the Microblaze instructions, we created the set of 30 applications described. We then used post-place and route models and XPA [4] to estimate the energy required to execute each application. The estimated energy values are graphed in Figure 2.28 against the ones count of the input array values. The figure also shows the linear approximation we make of the points. The lowest point of the line, at 194.7 nJ, corresponds to the least dense operands for the application given in Listing 2.3. This value is in fact only 0.4% less than the sum of the base energies of the instructions in the benchmark (which evaluates to 195.4 nJ using values in Table 2.3). The additional energy consumed, beyond the base energy cost of 195 nJ, is the accumulation of operand-caused energy variances of instructions in the benchmark (the $k \cdot EV(i)$ term in equation (2.6)).

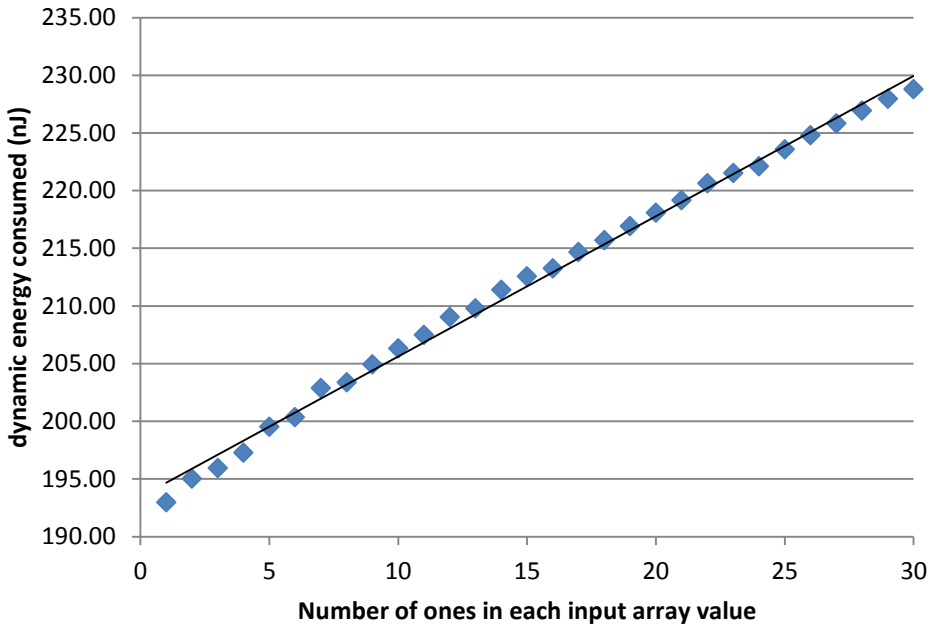


Figure 2.28 Relation between the energy consumed running an application and the one's count of its input values

The energy consumed by the programs in Figure 2.28, represented by $\sum E_i$, increases linearly with increase of the density of one's in the instruction operands values. Since the terms $\sum E_{base}(i,j)$ and $\sum EV(i)$ are constant, the k term in equation (2.6) is expressed as a linear function of the one's density as given in equation (2.7). The one's density of the operands of instruction i, is denoted by $OD(i)$ with the m and b terms denoting the slope and Y-intercept, respectively, of the linear fit.

$$\boldsymbol{k = m \cdot \big(OD(i)\big) + b} \qquad \textbf{(2.9)}$$

We can then substitute k into equation (2.6) and express the estimated energy consumed by a basic block of N instruction using equation (2.8) as the sum of the estimated energy consumed for each instruction in the basic block.

$$\boldsymbol{E_{est} = \sum_{i=1}^{N} E(i) = \sum_{i=1}^{N} E_{base}(i, i-1) + \sum_{i=1}^{N} (m \cdot OD(i) + b) \cdot \Delta E(i)} \qquad \textbf{(2.10)}$$

In order to use equation (2.10) with the results shown in Figure 2.28 to derive the values for *m* and *b* for a given processor, we can equate the estimated energy of the applications calculated using equation (2.8) with the accurate energy estimates found using the post-place and route model such as XPA [4]. The sum of base energies and energy variance for the instructions making up the application can be evaluated using the values in Table 2.4. The average one's density in the operands of each instruction ($OD(i)$ parameter) is found using the estimation and annotation tool described in the next chapter. This generates an over-determined system of 30 equations and two unknown variables *m* and *b*. By rewriting these equations, representing m in terms of b, the solution to this system can be visualized as given in Figure 2.29. This over-determined system converges to an approximate solution at $m = 0.016$ and $b = -0.061$ for the Microblaze processor. The values of *m* and *b* are added to the processor's OVBM.

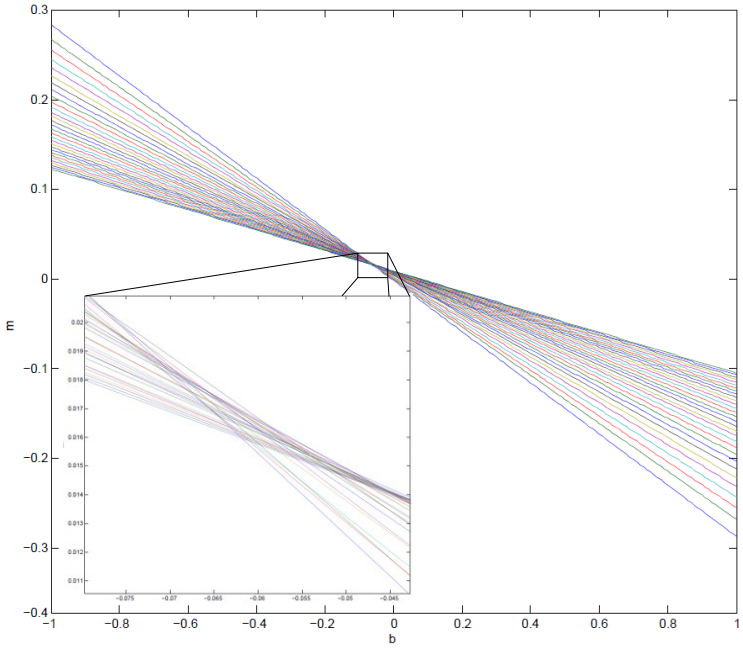


Figure 2.29 Graphs of the 30 linear equations used to evaluate the *m* and *b* parameters

2.2.3.2 Energy Impact of Alternating Bit Values

Instruction sequences like repeating shifts are frequently used by the Microblaze compiler and the energy impact of their operand value significantly differ from other instructions. The repeating shifts used by the Microblaze compiler operate on the same source and destination register. The first shift switches the processor from an unknown state determined by its preceding instruction. Therefore, its energy cost can be estimated using equations (2.6) and (2.9) discussed earlier. However, the second, and every other consecutive, shift instruction will require less switching of signals, since they do cause the processor to change its state. The most significant switching caused by each of these instructions is to shift the operand value once. Therefore, only where there are alterations in the bit values of the operand that switching will be required. That is, shifting 1010 will require switching all 4 bits, whereas shifting 0011 will require switching one bit, and therefore will consumed more energy.

This behavior is confirmed when the instructions of the 30 reference applications described in the previous subsection are replaced with a series of 31 shift instructions. The energy required to run these 30 applications is estimated using the post-place and route model. Furthermore, the energy required to run a refactored version of these applications with only the first shift instruction is also estimated. The difference of the energy required to execute the applications with the 31 shift instructions and those with one shift, approximates the energy required to run the 30 repeating shift instructions. From these estimates, the average energy per shift instruction is calculated. Knowing the values in the source array of each application, the average number of alternating bits in the operand value of the shift instructions is calculated. Figure 2.30 shows the energy dissipated by each repeated shift instruction on the Y-axis, and the average number of alternating bit values in the shift operand value on the X-axis. The fitted line can then be used to estimate the energy of

each repeating shift instruction as given in (2.9), where ABVC stands for the alternating bit value count and the m_{shift} and b_{shift} are the slope and Y-intercept of the line fitted to the points in Figure 2.30. The slope and Y-intercept parameters of the line we fitted to the data points are also added to the OVBM. For the Microblaze processor, the slope and the Y-intercept evaluate to 0.02, and 0.5 respectively. These parameters are also added to the OVBM, completing the list of parameters in the Microblaze OVBM as given in Table 2.5.

$$\boldsymbol{E_{repeating\ shift} = m_{shift} \cdot ABVC(i) + b_{shift}} \qquad \textbf{2.11)}$$

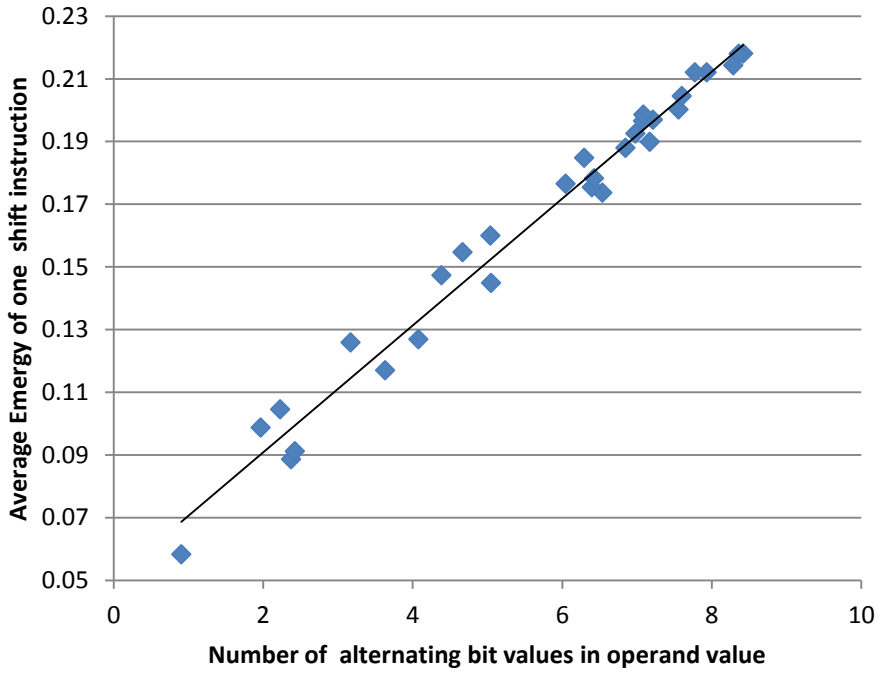


Figure 2.30 Energy consumed executing a repeating shift instruction v.s number of alternating bit values in its operand

Table 2.5 Complete list of the Microblaze OVBM parameters

Instruction	Base energy after instruction from class (nJ)			Max. instr. Energy Variance
	Arithmetic & Logic	*Memory*	*Shift*	
add	0.1147	0.4882	0.1608	1.0034
rsubk	0.3461	1.0352	0.7762	0.7872
mul	0.1233	0.4819	0.4019	0.9795
idiv	0.1850	0.5401	0.4419	0.7602
and	0.0892	0.5306	0.4213	0.6977
xori	0.3257	0.6345	0.5921	0.6977
cmp	0.1821	0.7108	0.5727	1.0456
nop	0.1343	0.4808	0.1959	0
lwi	0.7680	0.3536	0.9858	0.5310
swi	0.8159	0.4108	0.9761	0.2208
srl	0.1628	0.5550	0.1124	1.0782
sra	0.1571	0.5836	0.1899	1.0373
Operand Value Impact - Linear Fit Parameters				
m	0.016			
b	-0.061			
m_{shift}	0.02			
b_{shift}	0.5			

Chapter 3

Energy Estimation Tool

We developed a tool to utilize the OVBM and instruction-level processor energy models described in Chapter 2 to estimate the energy consumption of any application. The tool is designed to automatically analyze the source code of a given application, generate an annotated executable to collect run-time information. It then uses any instruction-level energy model to estimate the energy the processor would consume running the application. The tool works in two phases as shown in Figure 3.1. It prepares the inputs needed by the model in the first phase and uses the model equations to generate the energy report in the second. The following sections describe the operations performed by the tool in each phase to generate the energy estimates.

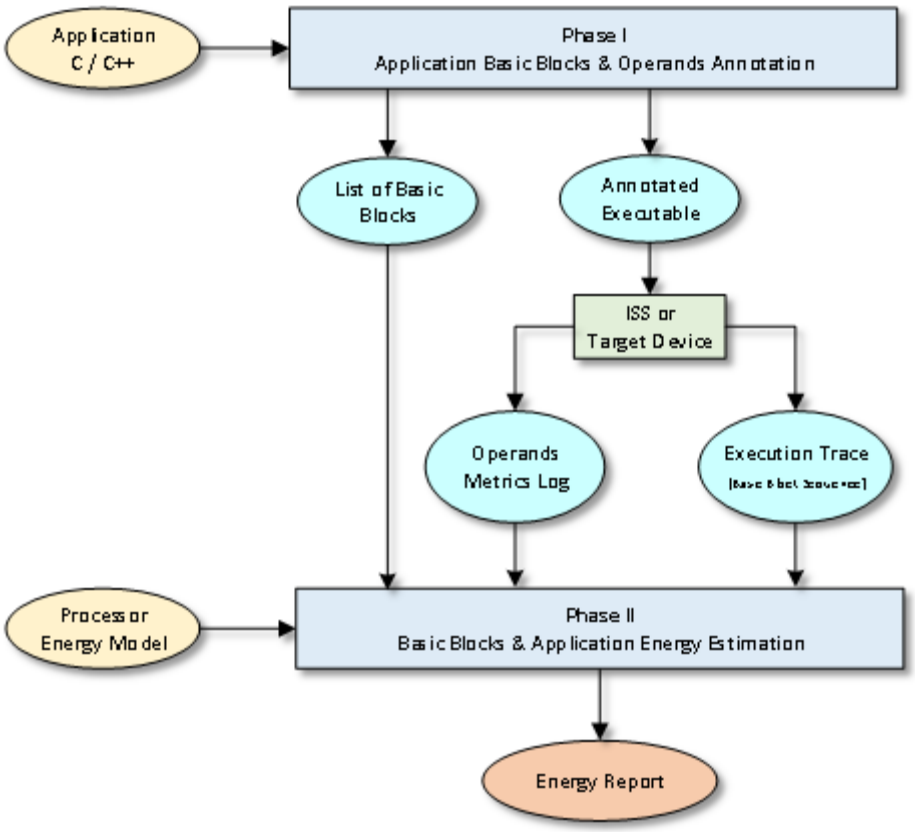


Figure 3.1 Proposed automated application analysis, annotation, and energy estimation tool

3.1 Phase 1

In the first phase, the tool is designed to generate the parameters needed by the OVBM model to estimate the energy required by a given application. It first identifies the basic blocks of the application, and the complete list of instructions in each basic block. It also generates an annotated executable that identifies the basic blocks that execute, the number of times they are executed, and the average one's density or number of alternating bit values in the operands of the different instructions. The annotations collect this information t at run time and transmit them to the host PC to be used in the energy estimation in the second phase. Figure 3.2 illustrates the process performed by the tool in the first phase as will be described in the following subsections.

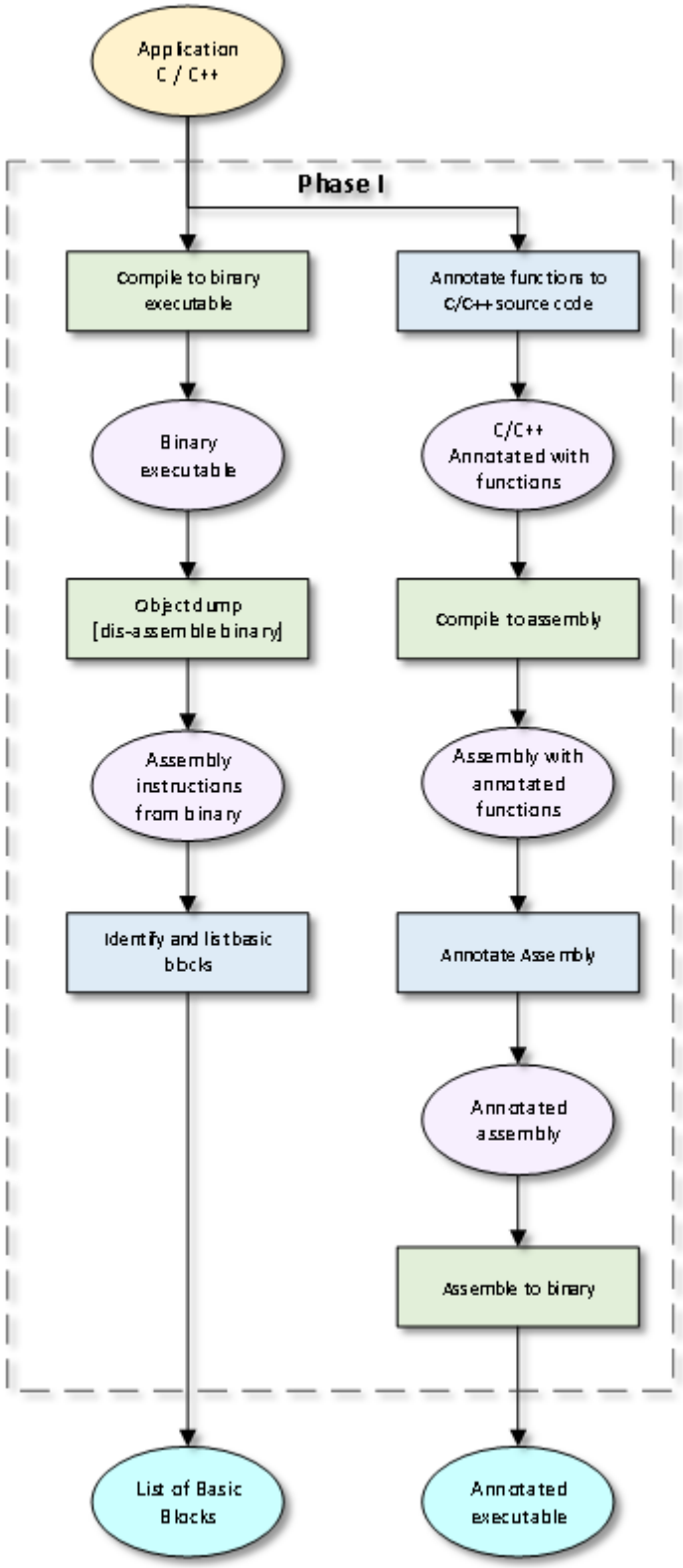


Figure 3.2 Phase I of the estimation tool. Identifying basic blocks and generating the annotated executable

3.1.1 Source Code Analysis and Basic Blocks Identification

The tool starts in phase 1 by generating the assembly code of the application to identify its basic blocks. A basic block is defined as a sequence of assembly instructions with one entry point and one exit point. This implies the instructions in each basic block always execute the exact same number of times in each application run. The tool lists the basic blocks and stores them in a file that is used as input in the second phase to estimate the energy of each using a processor energy model.

For the most accurate representation of the instructions in that will be executed in each basic block, the tool compiles the source code then it dis-assembles the binary file using an object-dump utility [29]. This is done because assemblers sometimes insert special instructions before generating the executable file. One example of such instructions is the Microblaze ***imm*** instruction. The encoded Microblaze immediate instructions contain 16-bit immediate field. However, the immediate value can be 32-bit value. When the assembler encounters an immediate instruction with a 32-bit immediate value, it pre-pends it with an ***imm*** instruction with the upper half of the 32-bit value. It also replaces the 32-bit value of the immediate instruction to the lower half of the value. This way, the ***imm*** instruction loads the upper half of the immediate values into the processor to be concatenated with the immediate value of the following instruction [35]. This is illustrated in the code snippet in Listing 3.1. It shows the instructions of the first basic block of the EncodeDC function from the JPEG encoder benchmark. The left column shows the instructions in the source code. The right column shows the instructions in the dis-assembled object file. The highlighted ***imm*** instructions were automatically inserted by the assembler. When estimating the energy of this basic block in the second phase, the tool will analyze the instructions read from the dis-assembled file to ensure it accounts for the energy of these instructions.

Listing 3.1 Instructions in Assembly Source File and dis-assembled Object File

Assembly from Source Code
EncodeDC:
Addik r1,r1,-44
swi r15,r1,0
swi r19,r1,40
addk r19,r1,r0
lwi r4,r0,input
lwi r3,r0,LastDC
rsubk r3,r3,r4
swi r3,r19,32
lwi r3,r0,input
swi r3,r0,LastDC
lwi r3,r19,32
bgei r3,$L53

Assembly from Object File
EncodeDC:
Addik r1, r1, -44
swi r15, r1, 0
swi r19, r1, 40
addk r19, r1, r0
imm 0
lwi r4, r0, -31444
imm 0
lwi r3, r0, -32484
rsubk r3, r3, r4
swi r3, r19, 32
imm 0
lwi r3, r0, -31444
imm 0
swi r3, r0, -32484
lwi r3, r19, 32
bgei r3, 20

In the sequence of assembly instructions read from the dis-assembled object file, the tool parses the instructions and identifies the boundaries of basic blocks by:

- Beginning of functions
- Return from subroutine instructions
- Branch and jump instructions
- Labels which used as target addresses of branch and jump instructions

When the executable file is generate, labels are removed, and displacement for branch instructions are calculated and inserted in the immediate field of the instruction. This can be seen in Listing 3.1, where the label of the last instruction `$L53` is replaced with the displacement value `20`. However, the object dump utility evaluates the address of the target location and appends it as a comment to the instruction in the dis-assembled file. This is illustrated in Listing 3.2. The end of basic block 2 is marked by the branch instruction in line 12. The displacement of the branch instruction is -32, i.e. 8 locations above the current location. The address of the target location, 240, is found appended to the instruction by the object-dump utility. Since this is an entry point to a sequence of instructions, line 4 is identified as the starting position of basic block 2. Furthermore, delayed instructions are identified and the instruction in the delay slot is included in the proper basic block. For instance, the return instruction in line 17 is delayed, line 18, *or r0, r0, r0* which implements a *nop*, is also included in basic block 3.

Listing 3.2 Identified basic blocks in dis-assembled instructions

Line #	Disassembled Instructions	Basic Block ID#
1	234: b0004000 imm 16384	1
2	238: 30600000 addik r3, r0, 0	
3	23c: f8730010 swi r3, r19, 16	
4	240: e8930018 lwi r4, r19, 24	2
5	244: e8730008 lwi r3, r19, 8	
6	248: 58641800 fadd r3, r4, r3	
7	24c: f8730014 swi r3, r19, 20	
8	250: e8930014 lwi r4, r19, 20	
9	254: b00042c8 imm 17096	
10	258: 30600000 addik r3, r0, 0	
11	25c: 58632210 fcmp.lt r3, r3, r4	
12	260: bc23ffe0 bnei r3, -32 // 240	
13	264: 10600000 addk r3, r0, r0	3
14	268: 10330000 addk r1, r19, r0	
15	26c: ea61001c lwi r19, r1, 28	
16	270: 30210020 addik r1, r1, 32	
17	274: b60f0008 rtsd r15, 8	
18	278: 80000000 or r0, r0, r0	

3.1.2 Source Code Annotations

The tool generates an annotated executable from the source code application to collect run time information such as the execution trace and operand values. This information is needed in the second phase as inputs to the OVBM to estimate the energy consumed by the application. The annotations are added to the compiled but not assembled source code, and not the dis-assembled application used to generate the list of basic blocks. This is because the dis-assembled application adds information to the assembly instructions. These include the binary encoding of the instruction,

and memory address as seen in Listing 3.2. Furthermore, instructions like the ***imm*** instruction described in the previous subsection can only be used by the assembler, and cannot be re-assembled.

The tool introduces three types of annotations to:

1. Record the execution sequence of the basic blocks.
2. Record the opcode of each instruction executed, as well as the values of their destination registers.
3. Calculate the average of the one's densities of all values used by each non-repeating shift instruction.
4. Calculate the average of alternating bit values for the repeating shift instructions.

To record run time data, the tool first annotates a logging function that logs the input value. The logging is done either through a serial port if the annotated application is intended to run on the target device or to a file if it is intended to run on an Instruction Set Simulator (ISS).

In order to record the execution trace, calls to the logging function are annotated in the assembly source code at the beginning of each basic block. In these annotations, the ID number of the basic block is passed as input the logging function. This way, as the annotated application is executed, a sequence basic block ID numbers will be logged tracing the execution path.

The second group of annotations used to record the instructions executing and their operand values, perform stack operations. They store in a stack in memory the opcode ID of the each instruction followed by the value stored in its destination register. These values are then processed by a function that is also annotated and called before the application terminates. In this function, the data stored in the stack is extracted and analyzed. As the instructions opcode ID numbers are read,

sequences of homogenous shift instructions are detected. The operand values corresponding to these instructions are analyzed, and the number of alternating bits is counted. For other instructions, the one's density of the operand values is calculated. The analysis produces two arrays containing the average density of ones in the operand values of each instruction type, and number of alternating ones in the operand value for each type of repeating shift instructions. The two arrays are then passed to the logging function to be stored for the second phase as values of the OD term in (2.7) and Alternating Bit Value Count (ABVC) term in (2.9).

These annotations are demonstrated in the segment of the annotated Dhrystone source code given in Listing 3.3. The instructions from the original source code are numbered and marked in bold. The listing shows parts of basic blocks 14 and 15 of the benchmark. At the beginning of each basic block, the annotated instructions set the value of register *r5* of the processor to the basic block ID number, then branches to an annotated logging function "trace". The trace function logs the basic block ID as part of the execution trace. Furthermore, following each original instruction, annotations push the instruction opcode and value of its destination operand into a stack with stack pointer stored in register r31. These values are processed to calculate the average ones densities and alternating bit values before the application terminates, to be used by the second phase.

Listing 3.3 Basic blocks number 14, and 15 from the Dhrystone benchmark

```
L12:
     # Exec trace annotations begin here
     addik     r5,r0,14 #Basic Block ID
     brlid     r15,trace
     nop     # Unfilled delay slot
     # Exec trace end here

 1   lbui     r3,r19,32
          addik     r30,r0,83     #
          swi  r30,r31,0x90000000 #Store opcode ID
          addik     r31,r31,4     #
          swi   r3,r31,0x90000000 #Store value
          addik     r31,r31,4     #
 2   sext8     r3,r3
          addik     r30,r0,130     #
          swi  r30,r31,0x90000000 #Store opcode ID
          addik     r31,r31,4     #
          swi     r3,r31,0x90000000  #Store value
          addik     r31,r31,4     #
. . .
 3   brlid     r15,Func_1
 4   nop          # Unfilled delay slot

     # Exec trace begin here
     addik     r5,r0,15 #Basic Block ID
     brlid     r15,trace
     nop     # Unfilled delay slot
     # Exec trace end here
. . .
```

3.2 Phase 2

The second phase of the estimation tool uses one of the instruction-level processor energy models and parameters obtained from the first phase to estimate the energy the application run would consume. It first uses one of the models to estimate the energy required by each basic block identified in the first phase. When using the first order model, equation (3.1) is applied using the estimated energy per instruction as given in Table 2.1. When using the second order model, equation (3.2) is applied using the parameters in Table 2.2, hence taking into account the inter-instruction energy effect.

$$\boldsymbol{E(BB) = \sum_{i=1}^{N} E(i)}\ 3 \qquad \textbf{(3.1)}$$

$$\boldsymbol{E(BB) = \sum_{i=1}^{N} E(i) \cdot (inst_i \neq int_{i-1}) + E(i) \cdot (r) \cdot (inst_i == inst_{i-1})} \qquad \textbf{(3.2)}$$

On the other hand, when using the proposed OVBM, equation (3.3) is used to estimate the energy of each instruction in each basic block using the base energy, maximum energy variance and linear parameters given in Table 2.5. It then estimates the energy consumed by each basic block of N instruction as the sum of the estimated energy of all its instructions as in (3.4).

$$\boldsymbol{E(i) = \begin{cases} E_{base}(i,j) + \left(m \cdot \left(OD(i)\right) + b\right) \cdot EV(i) & , if\ (i)\ is\ not\ a\ repeating\ shift \\ m_{shift} \cdot ABVC(i) + b_{shift} & , if\ (i)\ is\ a\ repeating\ shift \end{cases}} \qquad \textbf{(3.3)}$$

$$\boldsymbol{E(BB) = \sum_{i=1}^{N} E(i) = \sum_{i=1}^{N} E_{base}(i, i-1) + \sum_{i=1}^{N} (m \cdot OD(i) + b) \cdot \Delta E(i)} \qquad \textbf{(3.4)}$$

Finally, the tool counts the number of times each basic block is executed using the execution trace. Knowing the energy cost of each basic block, as estimated using any model, the tool calculates the total energy consumed by an application of M basic blocks using (3.5).

$$E_{app} = \sum_{j=1}^{M} E(BB(j)) \cdot Executions\_Count(BB(j)) \quad \textbf{(3.5)}$$

Furthermore, the use of instruction-level energy models makes it possible to generate granular energy profiles for the application. That is, besides the total energy consumption of an application, the execution trace along with the contents of each basic block can be used to profile the estimated energy consumed by each executing instruction, giving a very detailed profile. It is also possible to profile the execution by listing the energy consumed by each executing application. It can also calculate how much does each basic block contribute to the total energy consumption. Such details are very critical to guide power optimization efforts. Examples of these detailed profiles and some conclusions that can be derived from them are presented in the following chapter.

Chapter 4

Experimental Results

We developed an energy models for a Microblaze soft processor implementation on a Virtex5 FPGA using the proposed OVBM method and the state of the arts methods described in Chapter 2. We implemented the estimation tool described in Section 4 to automatically annotate and analyze applications targeted for a Microblaze processor. The automatic annotation tool was extended to use all three instruction level processor energy data models (first and second order models, as well as the proposed OVBM). All models characterize the dynamic energy consumption of a Microblaze soft processor implementation on a Virtex 5 FPGA, connected via a Local Memory Bus (LMB) to 64 kB block RAM, which stores the program and data of the application. The system clock is operating at a frequency of 125 MHz. The following sections will discuss the results from estimating the energy requirements for a set of 12 benchmark applications in terms of accuracy, speed, granularity, and estimation effort.

4.1 Accuracy

In order to compare the accuracy of the models, a set of 12 benchmarks was selected and the examined. The benchmarks include Dhrystone [6], an implementation of the quicksort algorithm, and the five functions of a JPEG encoder: Read BMP Block, *Discrete Cosine Transfer* (DCT), Quantize, Zigzag, and Huffman encode. Six variation of the quicksort implementation where tested. In each, the array being sorted is initialized with different values as described in Table 4.1. These benchmarks were selected because collectively they contain a great diversity of instructions

and operations with a great range of operand values. The Dhrystone benchmark is contains many integer arithmetic operations, string operations, memory accesses and logic decisions, presenting a good diverse set of instructions. Quicksort and Zigzag benchmarks consist of many memory access operations and logic decisions. Read-BMP block is made up of mostly memory access operations. DCT and Quantize benchmarks consist of many arithmetic and shift operations. Finally, the Huffman benchmark is made up of the most number of functions, with many memory access operations and logic decisions. As will be evident in the results, the state of the arts modeling techniques will produce good results for some of these benchmarks but not for others, unlike the proposed OVBM. This is critical to determine the robustness of the proposed technique in estimating different types of benchmarks with different types of operations.

Table 4.1 Quicksort benchmark applications examined

Application	Abbreviation	Notes
QuickSort V1	Qs1	Sorts an array of 10 integers between 10 and 19 in random order
QuickSort V2	Qs2	Sorts an array of 10 integers between 10 and 19 in ascending order
QuickSort V3	Qs3	Sorts an array of 10 integers between 10 and 19 in descending order
QuickSort V4	Qs4	Sorts an array of 10 integers between 10 and 10^6 in random order
QuickSort V5	Qs5	Sorts an array of 10 integers between 10^6 and 10^9 in random order
QuickSort V6	Qs6	Sorts an array of 50 integers between 1 and 500 in random order

4.1.1. Reference Estimates

The reference execution time and average dynamic power consumed by the processor for each benchmark were obtained using Xilinx iSim [32] and XPA [4]. The product of the execution time and the average power is the reference dynamic energy consumed by the benchmarks as shown in Table 4.2. Since XPA performs post place and route, transistor level simulations, these estimations

are highly accurate and can be used as reference to analyze and compare the accuracy of the different energy models.

Table 4.2 Benchmark energy estimates using XPA [4]

Application	Time (µs)	Power (mW)	Energy (mJ)
Dhrystone	39.35	33.35	1.31
Qs1	17.59	31.02	0.55
Qs2	14.01	31.57	0.44
Qs3	18.10	31.20	0.56
Qs4	22.18	31.61	0.70
Qs5	19.57	33.03	0.65
Qs6	164.20	33.78	5.55
ReadBMPBlock	251.61	39.96	10.05
DCT	166.68	30.84	5.14
Quantize	58.20	25.52	1.49
Zigzag	25.33	30.98	0.78
Huffman Encode	471.95	40.70	19.21
JPEG	973.77	37.66	36.67

4.1.2. State Based Models

Table 4.3 presents the dynamic energy estimations obtained using state based models created with different benchmarks as reference power. As described in Section 2.1.1, the state based model uses the average dynamic power consumed by one reference benchmark as the average power consumed by any application. The estimated energy is evaluated as the product of the execution time by the average power. Naturally, the error in estimating the benchmark used as reference power using the state based model is zero. If the developer is fortunate to choose a benchmark like the Dhrystone or first quicksort benchmark presented, the average estimation error will be reasonable for such a coarse approach. However, estimates using such an approach suffer greatly

when estimating the energy consumed by applications that contain a different distribution of instruction types and operations. This is seen in the poor estimations of the energy of the JPEG functions when compared to the reference energy values in Table 4.2. It is also important to note that the accuracy of estimation derived using state based models depend on the accuracy of the timing estimation used. In the results presented in Table 4.3, the accurate timing estimation used to evaluate the reference energy consumption in Table 4.2 is used with the state based models. However, very accurate execution time estimations may not be easy to obtain efficiently. The rest of the instruction-level models, including the proposed OVBM do not use the execution time as a parameter to estimating the energy of the program.

Table 4.3 Accuracy of state based energy model using different reference applications as compared to XPA [4] estimations in Table 1

	Estimated energy from state-base model and reference power estimate of:									
	Dhrystone		**Qs1**		**ReadBMP**		**Huffman Enc.**		**JPEG**	
Application	E (mJ)	Err	E (mJ)	Err	E (mJ)	Err	E (mJ)	Err	E (mJ)	Err
Dhrystone	1.31	0.0%	1.22	-7.0%	1.57	19.8%	1.60	22.0%	1.48	12.9%
Qs1	0.59	7.5%	0.55	0.0%	0.70	28.8%	0.72	31.2%	0.66	21.4%
Qs2	0.47	5.6%	0.43	-1.7%	0.56	26.6%	0.57	28.9%	0.53	19.3%
Qs3	0.60	6.9%	0.56	-0.6%	0.72	28.1%	0.74	30.4%	0.68	20.7%
Qs4	0.74	5.5%	0.69	-1.9%	0.89	26.4%	0.90	28.7%	0.84	19.1%
Qs5	0.65	1.0%	0.61	-6.1%	0.78	21.0%	0.80	23.2%	0.74	14.0%
Qs6	5.48	-1.3%	5.09	-8.2%	6.56	18.3%	6.68	20.5%	6.18	11.5%
ReadBMP	8.39	-16.5%	7.81	-22.4%	10.05	0.0%	10.24	1.9%	9.48	-5.8%
DCT	5.56	8.2%	5.17	0.6%	6.66	29.6%	6.78	32.0%	6.28	22.1%
Quantize	1.94	30.7%	1.81	21.6%	2.33	56.6%	2.37	59.5%	2.19	47.6%
Zigzag	0.84	7.7%	0.79	0.1%	1.01	29.0%	1.03	31.4%	0.95	21.6%
Huffman Enc.	15.74	-18.1%	14.64	-23.8%	18.86	-1.8%	19.21	0.0%	17.77	-7.5%
JPEG	32.48	-11.4%	30.21	-17.6%	38.91	6.1%	39.63	8.1%	36.67	0.0%
Average error		10.0%		9.3%		24.3%		26.5%		18.6%
Std. Deviation of error		8.3%		9.4%		13.6%		14.2%		10.7%

4.1.3. Instruction Level Models

lists the estimation results obtained using the instruction level models, first and second order state of the arts models, and the proposed OVBM. The first and second order models initially produced very large errors as shown in

. The energy for an instruction in the first order model is obtained by low-level simulations of the given instruction in an infinite loop as described in Section 2.1.2 and as was done in [2] [24] [25] [27]. Similarly, the energy of instructions and the inter-instruction energy effect in the second order model is obtained using low-level simulation the instructions in infinite loops as described in Section 2.1.3. Clearly, this technique does not produce accurate estimates of the energy cost of an instruction because of the false assumptions used in generating the models and explained in Section 2.1.2. As suggested in [8] [36], the models were calibrated using the Dhrystone benchmark estimation error to produce calibrated energy estimates. This calibration is performed by first finding the ratio of the estimated energy of the Dhrystone application using the first or second order model to the accurate estimate energy of the application using the accurate XPA tool. The energy estimates for the other applications are then divided by this ratio to obtain the calibrated energy estimates given in

. However, despite calibration, the first and second order models generated estimates with worst-case errors of up to 37.6%, and 38.5% respectively. The average errors were also high at 16% and 12.6%. Moreover, these models cannot be used with confidence because the standard deviation of the errors is high. These errors may significantly vary if a different benchmark was used to calculate the calibration ratio.

Table 4.4 Accuracy of different energy models with relative errors as compared to XPA [4] estimations in Table 1

Application	First order Model				Second order Model				OVBM	
	E (mJ)	**Err**	**E* (mJ)**	**Err**	**E (mJ)**	**Err**	**E* (mJ)**	**Err**	**E (mJ)**	**Err**
Dhrystone	3.6	171%	1.31	0.0%**	3.3	155%	1.31	0.0%**	1.30	-0.7%
Qs1	1.81	231%	0.58	6.1%	1.35	148%	0.53	-2.9%	0.57	4.2%
Qs2	1.70	285%	0.55	**23.6%**	1.07	142%	0.42	-5.1%	0.46	3.9%
Qs3	1.85	228%	0.59	5.1%	1.40	147%	0.55	-3.1%	0.58	3.7%
Qs4	2.28	225%	0.73	4.1%	1.70	142%	0.67	-5.1%	0.72	3.2%
Qs5	2.01	212%	0.65	-0.1%	1.50	131%	0.59	-9.3%	0.65	0.8%
Qs6	15.80	185%	5.07	-8.7%	12.63	128%	4.95	-10.7%	5.37	-3.2%
ReadBMP	24.6	145%	7.90	-21.4%	21.7	116%	8.50	**-15%**	8.82	**-12%**
DCT	18.2	253%	5.82	13.2%	18.2	253%	7.12	**38.5%**	4.96	-3.5%
Quantize	6.4	329%	2.04	**38%**	4.0	169%	1.57	5.4%	1.47	-0.9%
Zigzag	2.3	195%	0.74	-5.3%	2.3	194%	0.90	15.3%	0.78	-0.6%
Huffman Enc.	50.7	164%	16.3	-15.4%	47.7	148%	18.7	-2.7%	17.64	-8.2%
JPEG	102.2	179%	32.8	-10.7%	93.8	156%	36.8	0.3%	33.67	-8.2%
Average error		216%		12.6%		156%		9.5%		4.2%
Std. Deviation of error		51.6%		10.6%		35.0%		10.4%		3.5%

* Calibrated energy estimates. Calibration factor is derived using the error of estimating the Dhrystone benchmark.

** Zero estimation error because the estimation was generated using the reference energy of the Dhrystone benchmark.

The energy estimates generated using the approach presented in this paper are given in the last column in

, with the OVBM heading. The accuracy of this method is a significant improvement over the other methods examined in terms of average accuracy and estimation confidence; generating estimates with a maximum error of -12.3%. The average error is only 4.7%. The standard deviation of the

errors with our model is only 4.3%, which means that the dynamic energy estimates can be used with confidence for early design space exploration and software optimizations.

4.2 Speed

In addition to having a high average accuracy and confidence, the proposed estimation technique generates energy estimates much faster than the other accurate tools. Table 4.5 presents the time needed by our tool and XPA [4] to analyze each of the examined applications. Both tools were executed using PCs with Intel i7 quad-core processors and 16 GB of RAM. As described in Chapter 3, the tool is executed on a host machine in two phases. In the second phase, it uses metrics obtained from running an annotated executable that is generated in the first phase. In our examination, we used a Microblaze implementation on a Xilinx Virtex5 FPGA development board to run the annotated executable. The slowest operation needed by our tool is the transmitting the logs from the target device to the host through the serial JTAG - USB connector. However, the tool was designed to utilize the target device to analyze the operand values and send the minimum amount of data to the host through this bottleneck connection. The data transmitted consists only of a series of basic block ID numbers and the average operand density for each instruction as shown in Figure 3.1 Proposed automated application analysis, annotation, and energy estimation tool.

To estimate the energy consumed by the JPEG benchmark, which is the largest benchmark examines, the total execution time of the tool on the host machine did not exceed one 280ms. The time needed to run the annotated executable and transfer the logs to the host reached a maximum 1 minute and 45 seconds. This gives a total worse case time of about 1 minute and 45 seconds. Compared to 10 hours and 38 minutes required to obtain the reference estimate using XPA, illustrates a speed up of 3 orders of magnitude.

Table 4.5 Execution time of OVBM-based estimation tool compared to Post-place-and-route simulation (XPA)

Application	OVBM Tool (Seconds)			XPA	
	Host	Target	Total	(Seconds)	(Minutes)
Dhrystone	0.03	7.49	7.53	4320	72
Qs6	0.01	23.08	23.09	8820	147
ReadBMPBlock	0.21	5.88	6.08	12120	202
DCT	0.03	10.85	10.88	8880	148
Quantize	0.01	8.40	8.41	5100	85
Zigzag	0.01	4.41	4.42	3900	65
Huffman Encode	0.07	65.04	65.11	20400	340
JPEG	0.28	104.24	104.52	38280	638

4.3 Estimation Granularity

The estimation tool as described in Chapter 3 is capable of generating fine-grained energy estimations using the proposed OVBM. The tool produces the execution trace of the examined benchmark as a sequence of executing basic blocks. It also uses the OVBM to estimate the energy consumed by each basic block. Combined, the software developer would be able to trace the execution of his application alongside the estimated energy consumed by each basic block. This level of granularity can guide software developers in their effort to optimize the energy consumption of their application. On the other hand, accurate estimation tools like XPA only produce the total estimated energy for a given application execution.

Figure 4.1 through Figure 4.24 provide a graphical presentation of the estimated energy generated by the tool. On the X-axis of each figure is the application execution trace, i.e. the sequence of

basic block ID numbers that execute. On the Y-axis of the odd-numbered figures, the dynamic energy consumed by each executed basic block. The Y-axis of the even-numbered figures displays the cumulative energy consumed by the application. The final point on the even-numbered figures is the estimated energy consumed by the application as given in

. Furthermore, since all the instructions in each basic block are known from the output of the first phase of the tool, the granularity of the output can be further expanded to instruction-level granularity. On the other hand, accurate estimation tools like XPA, produce a single value for the average dynamic power consumed, which can be used to calculate the single energy estimate as given in Table 4.2.

Figure 4.1 presents graphically the estimated energy of each basic block executed by the Dhrystone benchmark as generated by the tool. The X-axis represents the execution trace; i.e. the sequence of executing basic block ID numbers. The Y-axis represents the estimated energy of the corresponding basic block using the OVBM as evaluated in the second phase. Figure 4.2 presents the cumulative energy consumed by the application up to and including the execution of the basic block on the X-axis. Combined, the two figures show that basic blocks 42, 43, 46, 48, and 49 are the most frequently executed basic blocks, responsible for approximately two thirds of the total dynamic energy consumed. It also shows basic blocks like 65, and 12 which consume more energy than other basic blocks execute with much less frequency, responsible for only about 5% of the total energy consumed. Such detailed analysis is very simple using the generated report using the proposed technique, and very difficult to perform using the reference accurate tools. This is demonstrated in the other energy reports for the benchmarks examined.

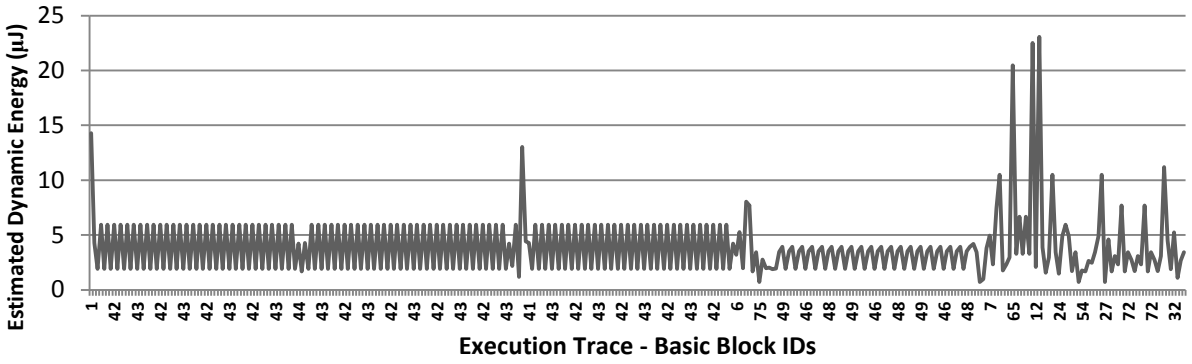


Figure 4.1 Estimated energy of all executed basic blocks in the Dhrystone benchmark

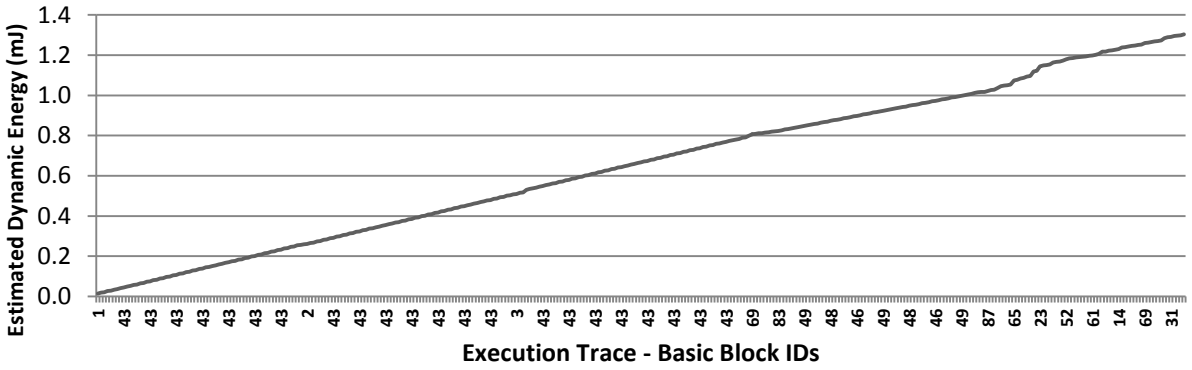


Figure 4.2 Cumulative estimated energy for the Dhrystone benchmark execution

Figure 4.3 presents graphically the estimated energy of each basic block executed by the QuickSort V1 benchmark as generated by the tool. The X-axis represents the execution trace. The Y-axis represents the estimated energy of the corresponding basic block using the OVBM. Figure 4.4 presents the cumulative energy consumed by the application up to and including the execution of the basic block on the X-axis. Combined, the graphs profile the energy consumed while the quicksort algorithm is applied to an array of 10 integers with values between 10 and 19 initialized in a random order. Therefore, this quick sort implementation has the lowest execution time and minimum energy consumption among the different versions of the QuickSort benchmarks presented.

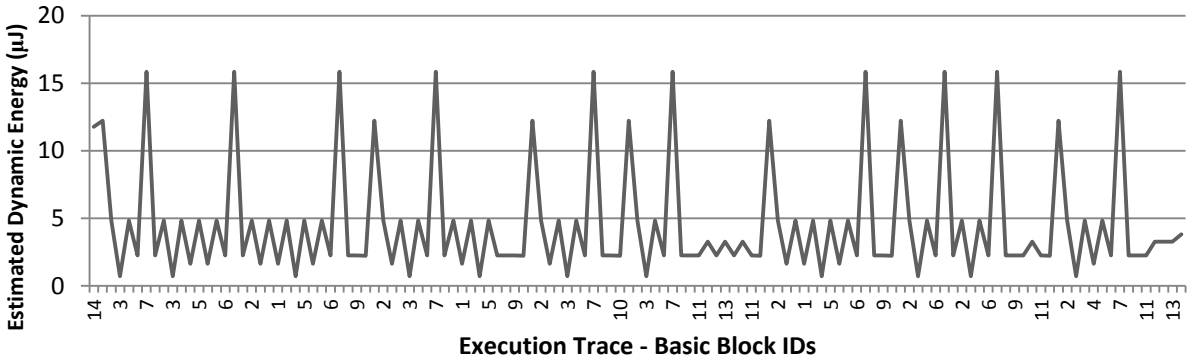


Figure 4.3 Estimated energy of all executed basic blocks in the Quicksort V1 benchmark

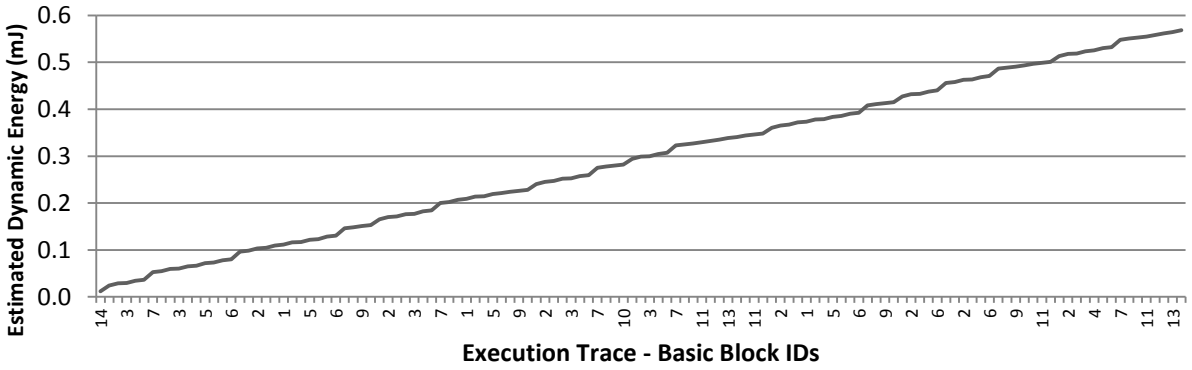


Figure 4.4 Cumulative estimated energy for the Quicksort V1 benchmark execution

Figure 4.5 presents graphically the estimated energy of each basic block executed by the QuickSort V2 benchmark as generated by the tool. The X-axis represents the execution trace. The Y-axis represents the estimated energy of the corresponding basic block using the OVBM. Figure 4.6 presents the cumulative energy consumed by the application up to and including the execution of the basic block on the X-axis. Combined, the graphs profile the energy consumed while the quicksort algorithm is applied to an array of 10 integers with values between 10 and 19 initialized in an ascending order. Therefore, this quick sort implementation has lowest execution time and minimum energy consumption among the different versions of the QuickSort benchmarks presented.

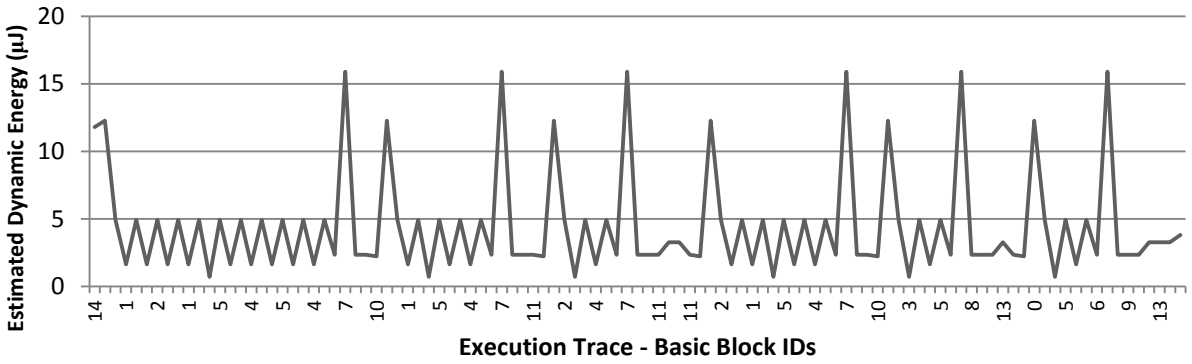


Figure 4.5 Estimated energy of all executed basic blocks in the Quicksort V2 benchmark

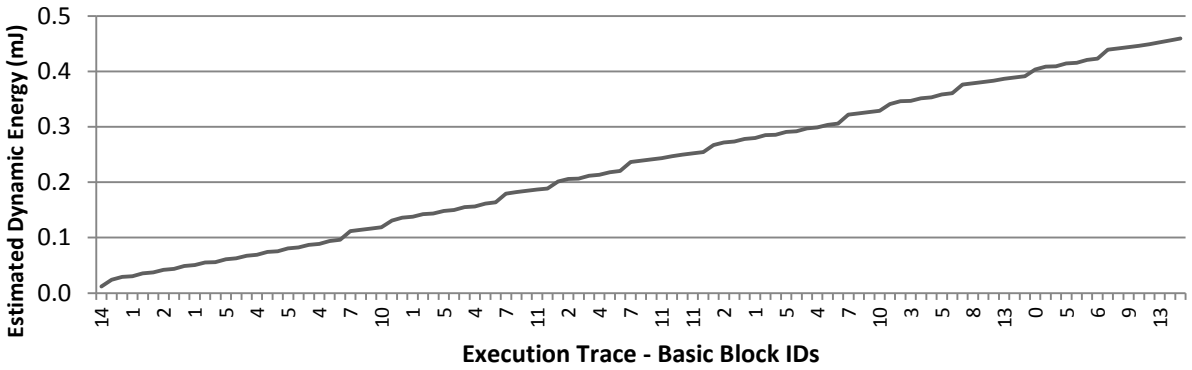


Figure 4.6 Cumulative estimated energy for the Quicksort V2 benchmark execution

Figure 4.7 presents graphically the estimated energy of each basic block executed by the QuickSort V3 benchmark as generated by the tool. The X-axis represents the execution trace. The Y-axis represents the estimated energy of the corresponding basic block using the OVBM. Figure 4.8 presents the cumulative energy consumed by the application up to and including the execution of the basic block on the X-axis. Combined, the graphs profile the energy consumed while the quicksort algorithm is applied to an array of 10 integers with values between 10 and 19 initialized in a descending order.

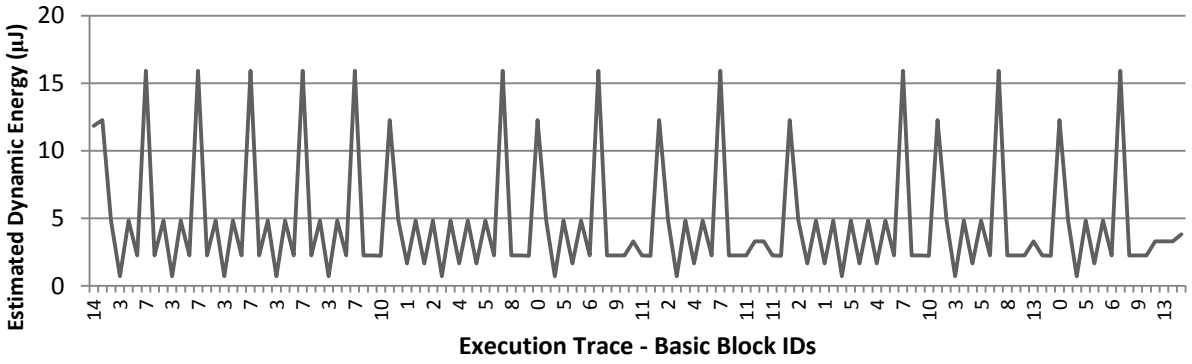


Figure 4.7 Estimated energy of all executed basic blocks in the Quicksort V3 benchmark

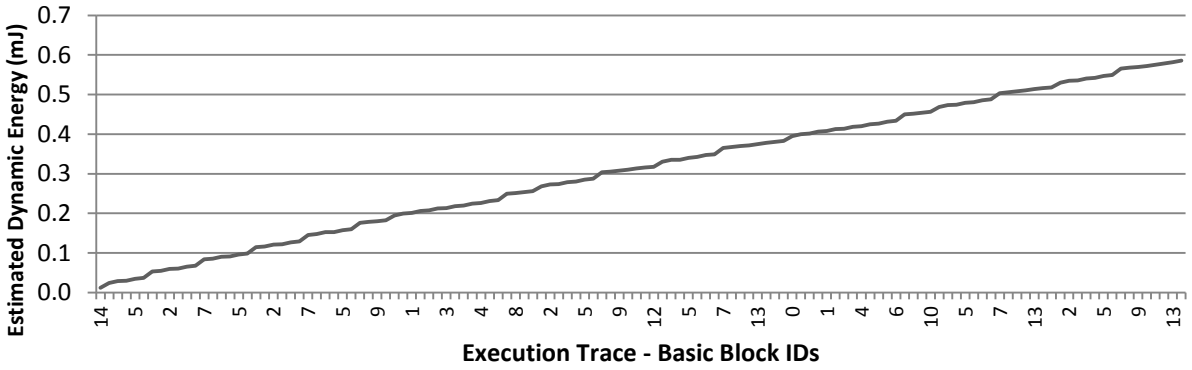


Figure 4.8 Cumulative estimated energy for the Quicksort V3 benchmark execution

Figure 4.9 presents graphically the estimated energy of each basic block executed by the QuickSort V4 benchmark as generated by the tool. The X-axis represents the execution trace. The Y-axis represents the estimated energy of the corresponding basic block using the OVBM. Figure 4.10 presents the cumulative energy consumed by the application up to and including the execution of the basic block on the X-axis. Combined, the graphs profile the energy consumed while the quicksort algorithm is applied to an array of 10 integers with values between 10 and 10^9 initialized in a random order. Because the values of the array have higher ones density than the previously presented QuickSort benchmarks, the estimated energy of the instructions loading, comparing and

storing these values is higher in this benchmark than in the previous QuickSort benchmarks. As a result, the energy of the basic blocks are also higher than in the previous QuickSort benchmarks.

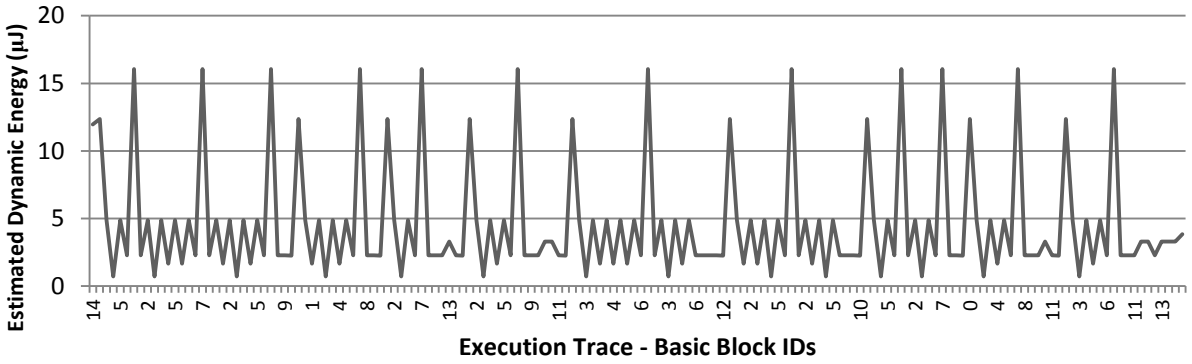


Figure 4.9 Estimated energy of all executed basic blocks in the Quicksort V4 benchmark

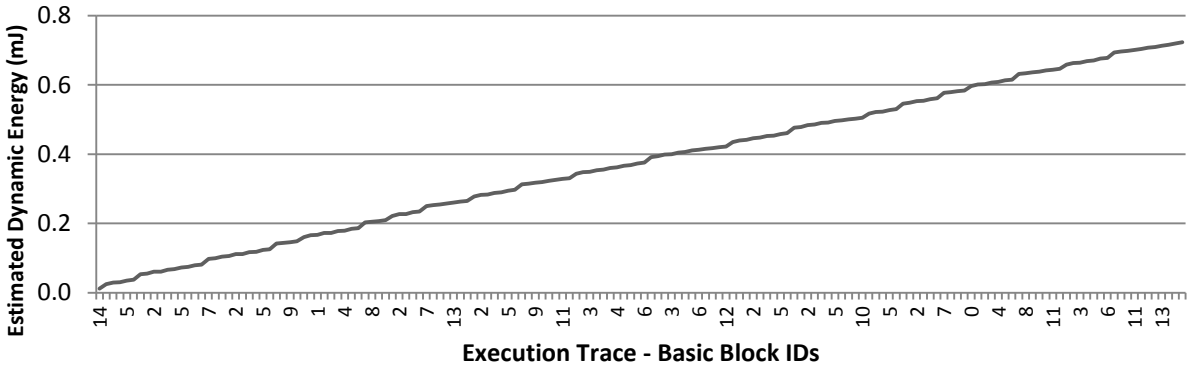


Figure 4.10 Cumulative estimated energy for the Quicksort V4 benchmark execution

Figure 4.11 presents graphically the estimated energy of each basic block executed by the QuickSort V5 benchmark as generated by the tool. The X-axis represents the execution trace. The Y-axis represents the estimated energy of the corresponding basic block using the OVBM. Figure 4.12 presents the cumulative energy consumed by the application up to and including the execution of the basic block on the X-axis. Combined, the graphs profile the energy consumed

while the quicksort algorithm is applied to an array of 10 integers with values between 10^6 and 10^9 initialized in a random order. Because the values of the array have the highest ones density among the presented QuickSort benchmarks, the estimated energy of the instructions loading, comparing and storing these values is higher in this benchmark than in the other QuickSort benchmarks. As a result, the energy of the basic blocks estimated are the highest among the presented QuickSort benchmarks.

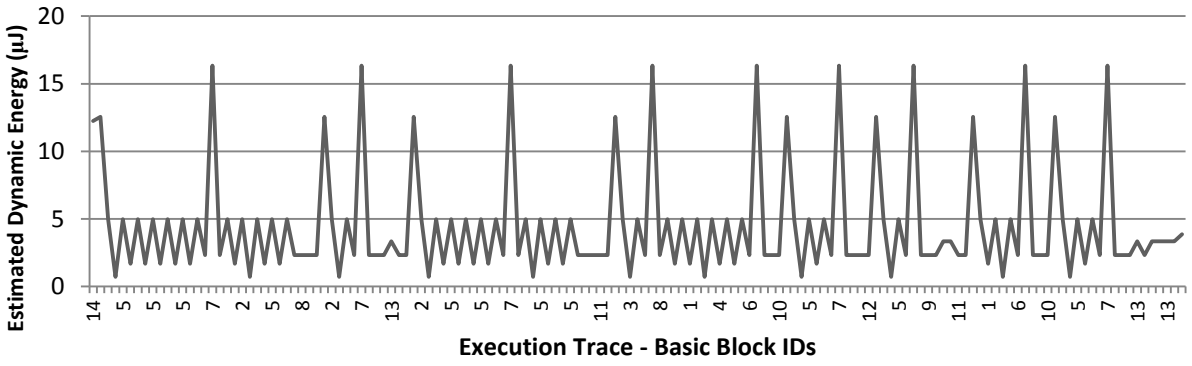


Figure 4.11 Estimated energy of all executed basic blocks in the Quicksort V5 benchmark

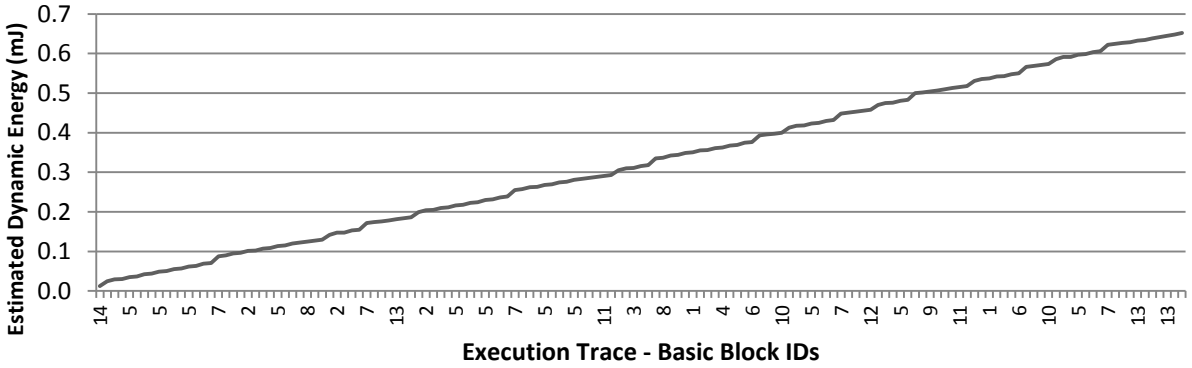


Figure 4.12 Cumulative estimated energy for the Quicksort V5 benchmark execution

Figure 4.13 presents graphically the estimated energy of each basic block executed by the QuickSort V6 benchmark as generated by the tool. The X-axis represents the execution trace. The Y-axis represents the estimated energy of the corresponding basic block using the OVBM. Figure 4.14 presents the cumulative energy consumed by the application up to and including the execution of the basic block on the X-axis. Combined, the graphs profile the energy consumed while the quicksort algorithm is applied to an array of 50 integers with values between 1 and 500 initialized in a random order. Therefore, the execution time and total energy consumption is the highest for this QuickSort benchmark among the other QuickSort benchmarks presented.

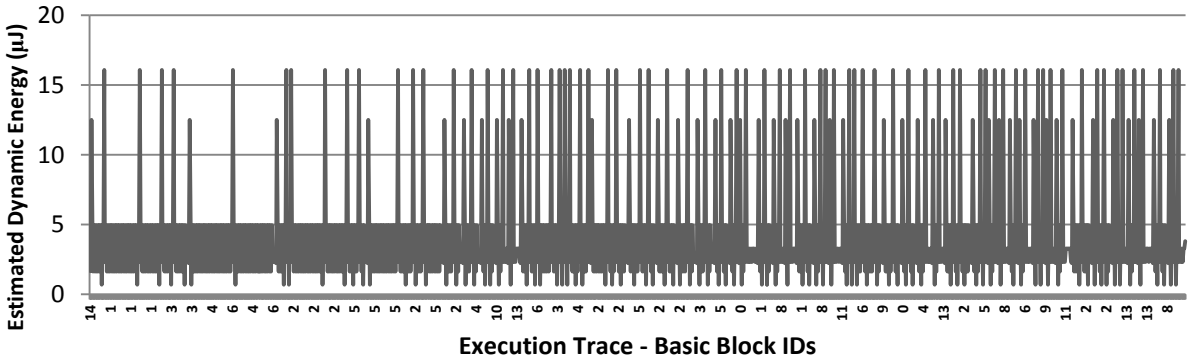


Figure 4.13 Estimated energy of all executed basic blocks in the Quicksort V6 benchmark

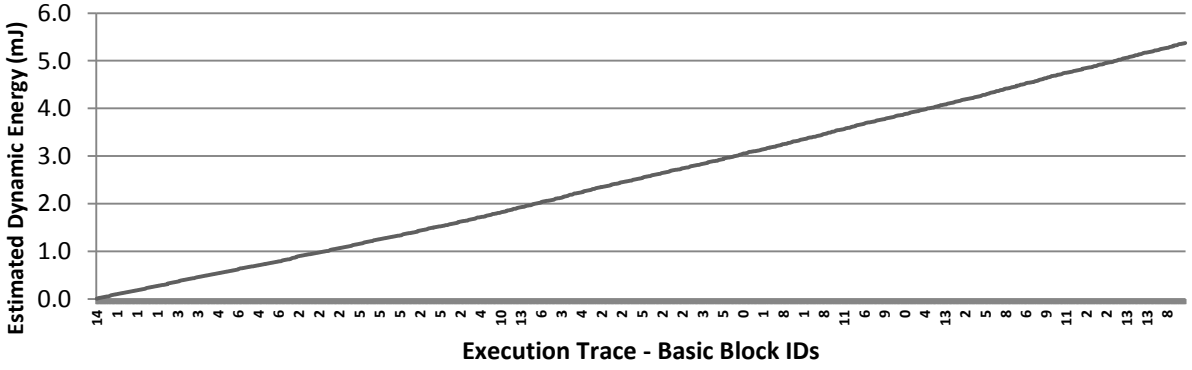


Figure 4.14 Cumulative estimated energy for the Quicksort V6 benchmark execution

Figure 4.15 presents graphically the estimated energy of each basic block executed by the ReadBMPBlock benchmark as generated by the tool. The X-axis represents the execution trace. The Y-axis represents the estimated energy of the corresponding basic block using the OVBM. Figure 4.16 presents the cumulative energy consumed by the application up to and including the execution of the basic block on the X-axis. These graphs clearly illustrate the execution of many basic blocks in a loop of 8 iterations that consume approximately 90% of the total energy consumed by the benchmark. The software developer would therefore be recommender to focus power optimization effort on these repeating basic blocks.

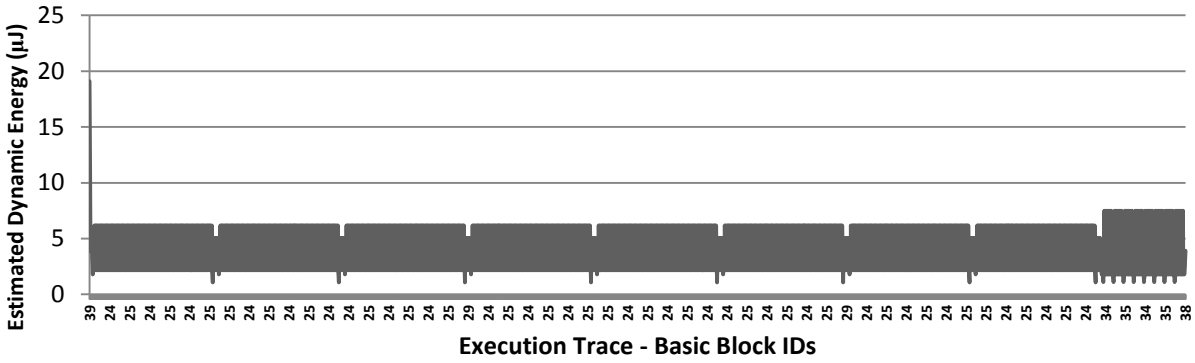


Figure 4.15 Estimated energy of all executed basic blocks in the ReadBMPBlock benchmark

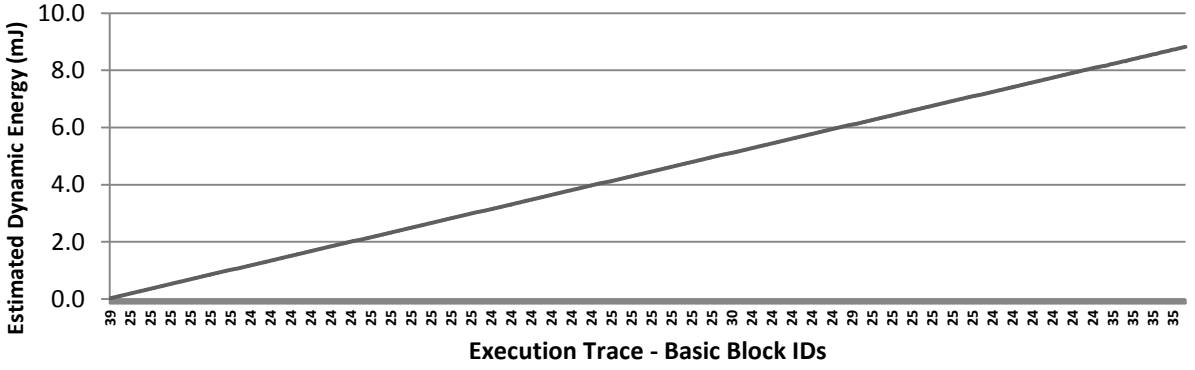


Figure 4.16 Cumulative estimated energy for the ReadBMPBlock benchmark execution

Figure 4.17 presents graphically the estimated energy of each basic block executed by the DCT benchmark as generated by the tool. The X-axis represents the execution trace. The Y-axis represents the estimated energy of the corresponding basic block using the OVBM. Figure 4.18 presents the cumulative energy consumed by the application up to and including the execution of the basic block on the X-axis. The energy profile generated demonstrates that approximately half the energy consumed by this benchmark is a result of executing basic blocks 5 and 8. Unlike in most benchmarks, most of the energy consumed resulted from executing these two basic blocks, which execute less frequently than the other basic blocks.

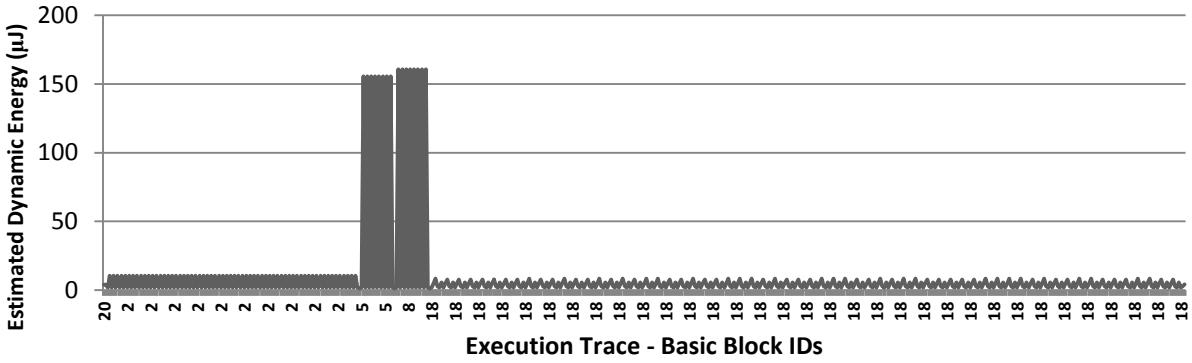


Figure 4.17 Estimated energy of all executed basic blocks in the DCT benchmark

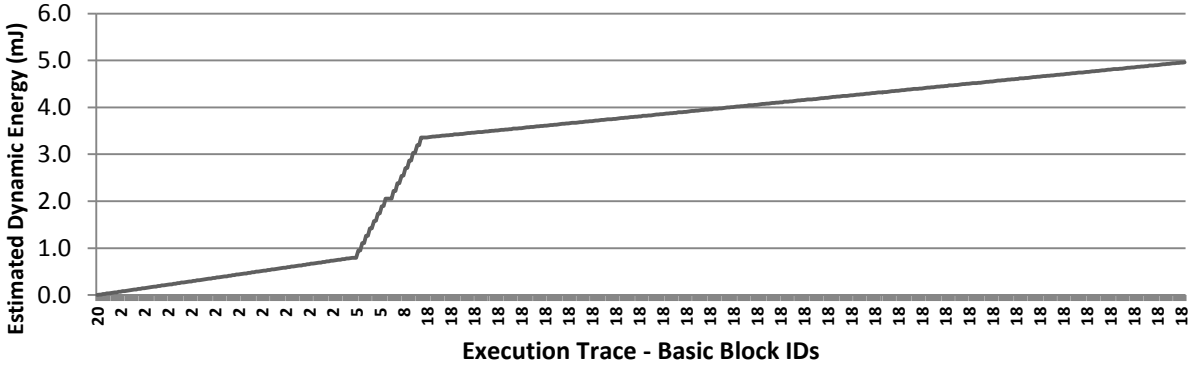


Figure 4.18 Cumulative estimated energy for the DCT benchmark execution

Figure 4.19 presents graphically the estimated energy of each basic block executed by the Quantize benchmark as generated by the tool. The X-axis represents the execution trace. The Y-axis represents the estimated energy of the corresponding basic block using the OVBM. Figure 4.20 presents the cumulative energy consumed by the application up to and including the execution of the basic block on the X-axis. These graphs demonstrate a steady energy consumption by the Quantize benchmark. Basic block 7 clearly consuming the most amount of energy.

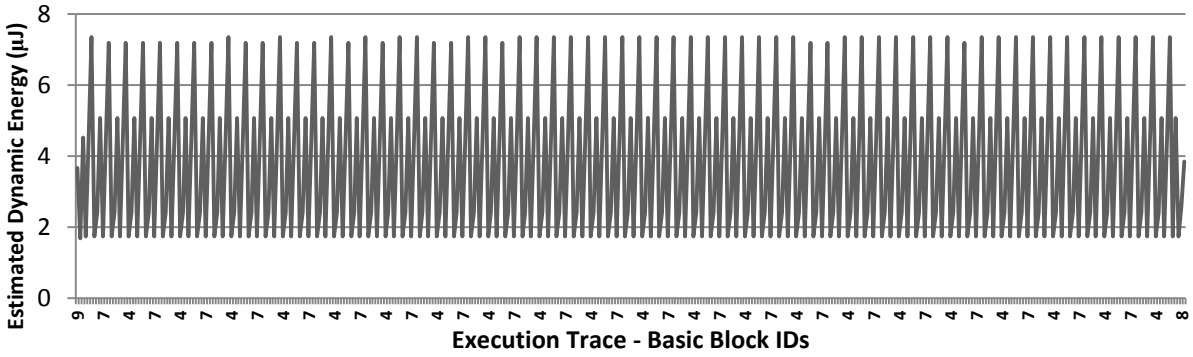


Figure 4.19 Estimated energy of all executed basic blocks in the Quantize benchmark

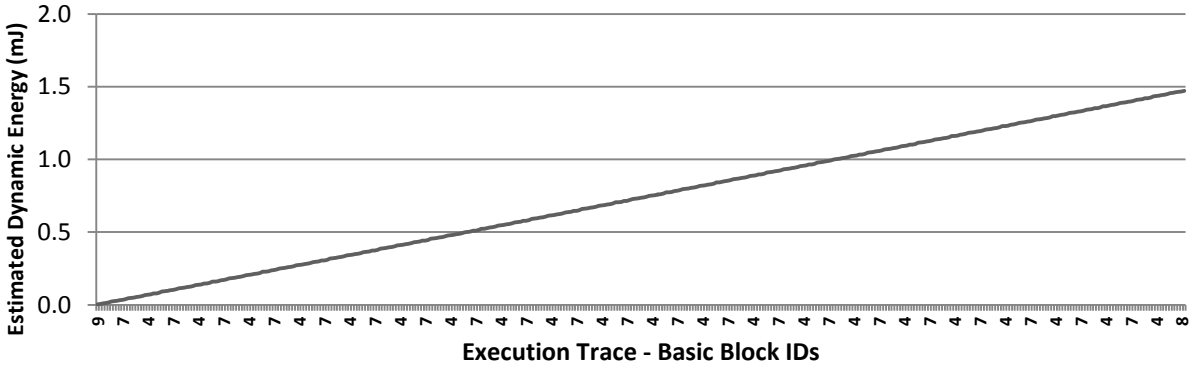


Figure 4.20 Cumulative estimated energy for the Quantize benchmark execution

Figure 4.21 presents graphically the estimated energy of each basic block executed by the Zigzag benchmark as generated by the tool. The X-axis represents the execution trace. The Y-axis

represents the estimated energy of the corresponding basic block using the OVBM. Figure 4.22 presents the cumulative energy consumed by the application up to and including the execution of the basic block on the X-axis. These graphs demonstrate the simple execution and steady increase in energy consumption of the Zigzag benchmark.

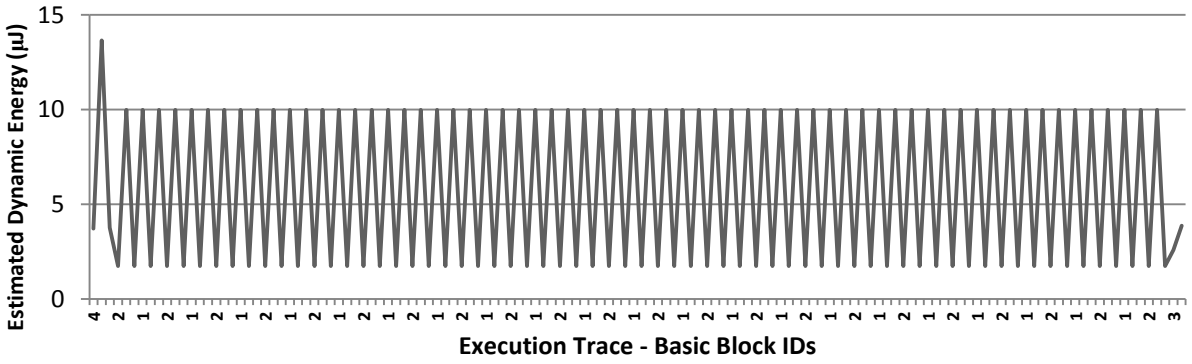


Figure 4.21 Estimated energy of all executed basic blocks in the Zigzag benchmark

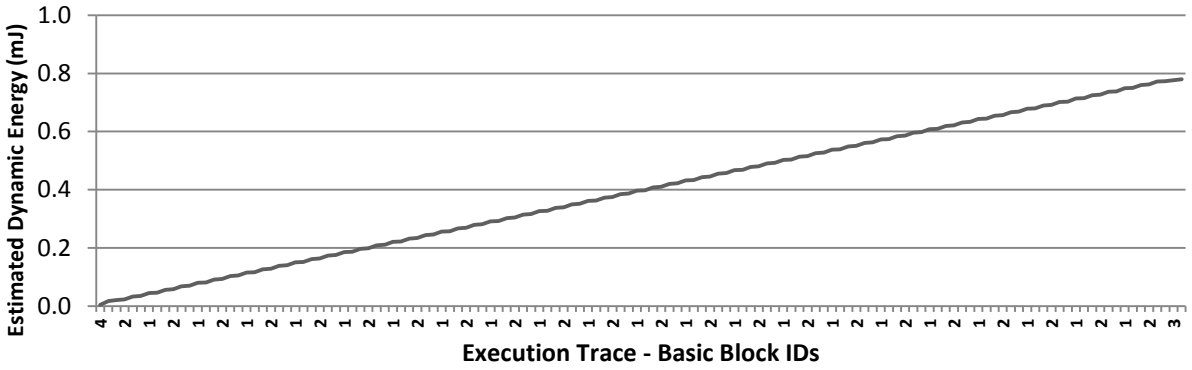


Figure 4.22 Cumulative estimated energy for the Zigzag benchmark execution

Figure 4.23 presents graphically the estimated energy of each basic block executed by the Huffman Encoder benchmark as generated by the tool. The X-axis represents the execution trace. The Y-axis represents the estimated energy of the corresponding basic block using the OVBM. Figure 4.24 presents the cumulative energy consumed by the application up to and including the execution of the basic block on the X-axis. These graphs demonstrate the complexity of this

benchmark as compared to the other benchmarks presented. However, the total energy estimated using the proposed tool was only 8% below the accurate estimate found using the accurate low-level tools.

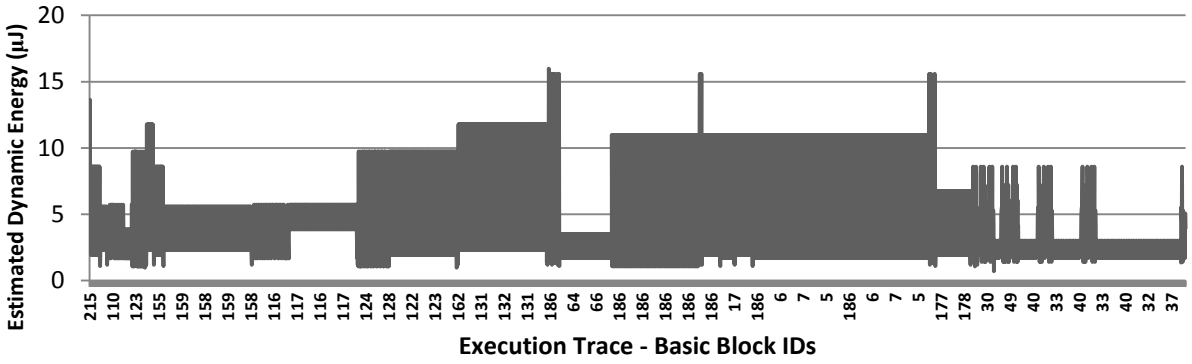


Figure 4.23 Estimated energy of all executed basic blocks in the Huffman Encoder benchmark

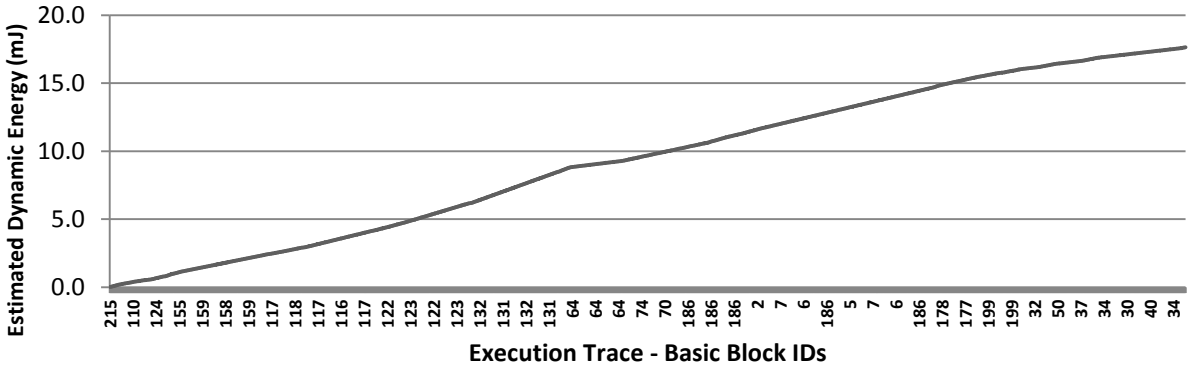


Figure 4.24 Cumulative estimated energy for the Huffman Encoder benchmark execution

4.4 Estimation Effort

The increase in average accuracy of the instruction model requires higher characterization effort. As was described in Chapter 2, creating a state based model requires estimating the average power required to run a single benchmark, like Dhrystone. The effort required to create the first and

second order models is proportional to the number of instructions in the processor instruction set. Given an instruction set of n instructions, first and second order models require estimating the energy consumed by n and n^2 instruction benchmarking applications, respectively. The effort required to develop an OVBM model depends on n and on the size of the reference loop used to generate the LBEPs. Given a reference loop of m instructions, $n \times m$ benchmarking applications are required to generate the base energy costs. A second set of $n \times m$ applications are required to generate the maximum energy variance. To model the energy impact of operands values, 30 applications are used to model the impact of operand density and 60 applications are used to model the impact of alternating bit values. Therefore, the energy required to run a total of $2(n \times m) + 90$ benchmarking applications is needed to develop an OVBM. It is important to note however the process of generating these applications and simulating them using reference low-level models and tools is easy to automate. In fact, we used basic scripts running on three computers with multi-core processors to continuously generate the benchmarking applications and run these simulations in parallel.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In this study, we presented a novel dynamic energy modeling technique for soft processors in FPGA based on the operand values of instructions. We compared it to accurate but slow low-level models, efficient but inaccurate high-level state based models as well as other instruction level modelling techniques. We identified a critical weakness in the conventional method used to characterize the energy of instructions in instruction level models, and proposed an alternative characterization technique. We also proved the energy consumption of instruction in soft processors is greatly affected by the operand values. We identified two structures of instructions that depend on two different properties of the operand values. The Energy of heterogeneous sequences of instructions depend on the density of ones in the operand values. On the other hand, the energy of homogeneous sequences of shift instructions depend on the number of alternating bit values in the operand. The proposed technique utilized the novel instruction characterization technique and it accounted for inter-instruction effects and operand values to estimate the energy of instructions with high accuracy and efficiency.

Furthermore, we present an automated tool designed to generate detailed energy reports for any given execution of an application using any instruction-level energy model. The tool operates in two phases. In the first phase, it analyzes the basic blocks and annotates the instructions of an application to generate an annotated executable. Using this executable, the user can run the

application as he would the original application, while the annotations trace the execution and operand values. Once the execution is complete, other annotations analyze the operand values used by all executed instructions, calculating the ones densities and alternating bit values. The estimation tool then uses this information in the second phase to generate reports, detailing the estimated dynamic energy consumed by the application.

We validate the proposed approach by modelling the Microblaze soft processor using this technique as well as a low-level model, high-level state based model, and two state of the arts instruction-level models. We used all these models to estimate the energy consumed by a set of 12 benchmarks with diverse distributions of operations, instructions, and operand values. By comparing the results, we prove the proposed technique is more than twice as accurate as state of the arts instruction-level techniques, and more than three times as accurate as high-level models. More importantly, we prove that the proposed technique can be used with great confidence with any type of application without calibrating the results. This approach also presents other advantages to software developers and system designers that are not possible using the low-level energy models or physical measurements. The granularity of the estimates can guide software optimization efforts for energy consumption. Furthermore, the presented approach generates estimates in up to 3 orders of magnitude faster than the accurate low-level estimation tools. Therefore, the proposed model can be used for thorough and extensive early design space exploration.

5.2 Future Work

In this thesis we presented an energy modeling and estimation technique for soft processors, and demonstrated its accuracy, efficiency, and granularity for estimating the energy consumed by a soft processor core. In the future, we aim to validate this approach for multiple processor systems

in FPGA. We also intend to incorporate the energy model and estimation tool presented in this thesis in a suite of early performance metrics estimation tools for embedded systems. Therefore, we wish to expand the estimation tool to estimate the timing performance using instruction-level performance estimation models such as the one presented in [37]. Being able to estimate the execution time, the tool would be able to generate both dynamic and static energy and power reports for the soft processors.

Furthermore, we aim to incorporate energy models of other system components to the estimation tool to estimate system-level energy consumption. Such models include main memory, system buses, clock trees, and other system controllers. In addition, further work can be done to expand the modeling technique to estimate the energy of soft processors with different cache configurations.

References

[1] N. Bansal, K. Lahiri, a. Raghunathan and S. T. Chakradhar, "Power Monitors: A Framework for System-Level Power Estimation Using Heterogeneous Power Models," in *18th International Conference on VLSI Design*, 2005.

[2] A. Sinha and A. P. Chandrakasan, "JouleTrack-a web based tool for software energy profiling," in *Design Automation Conference, 2001. Proceedings*, 2001.

[3] A. Krishnaswamy and R. Gupta, "Dynamic Coalescing for 16-bit Instructions," *ACM Transactions in Embedded Computing Systems - TECS,* vol. 4, pp. 3-37, 2004.

[4] Xilinx Inc., "XPower Analyzer," 2012. [Online]. Available: http://www.xilinx.com/products/design_tools/logic_design/verification/xpower_an.htm.

[5] Synopsys, 2012. [Online]. Available: http://www.synopsys.com/Tools/Implementation/RTLSynthesis/Pages/PowerCompiler.aspx.

[6] R. P. Weicker, "Dhrystone: a synthetic systems programming benchmark," *Communications of the ACM,* vol. 27, pp. 1013-1030, 1984.

[7] Xilinx Inc., "LogiCORE IP MicroBlaze micro controller system," 2012.

[8] G. K. Wallace, "The JPEG still picture compression standard," *Consumer Electronics, IEEE Transactions,* vol. 38, pp. xviii-xxxiv, 1992.

[9] Xilinx, "XPower Estimator," 2011.

[10] L. Qi, B. Guo, Y. Shen, J. Wang, Y. Wu and Y. Liu, "An embedded software power model based on algorithm complexity using back-propagation neural networks," in *reen Computing and Communications (GreenCom), 2010 IEEE/ACM Int'l Conference on & Int'l Conference on Cyber, Physical and Social Computing (CPSCom)*, 2010.

[11] U. B. Correa, L. Lamb, L. Carro, L. Brisolara and J. Mattos, "Towards estimating physical properties of embedded systems using software quality metrics," in *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference*, 2010.

[12] C. Bachmann, A. Genser, C. Steger, R. Weiss and J. Haid, "Automated power characterization for run-time power emulation of SoC designs," in *Digital System Design: Architectures, Methods and Tools (DSD), 2010 13th Euromicro Conference*, 2010.

[13] J. Haid, C. Kaefer, C. Steger and R. Weiss, "Run-time energy estimation in system-on-a-chip designs," in *Design Automation Conference, 2003. Proceedings of the ASP-DAC 2003. Asia and South Pacific*, 2003.

[14] A. Krieg, C. Bachmann, J. Grinschgl, C. Steger, R. Weiss and J. Haid, "Accelerating early design phase differential power analysis using power emulation techniques," in *ardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium*, 2011.

[15] M. Rogers-Vallee, M. Cantin, L. Moss and G. Bois, "IP characterization methodology for fast and accurate power consumption estimation at transactional level model," in *Computer Design (ICCD), 2010 IEEE International Conference*, 2010.

[16] 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*, 2009.

[17] S. Gurumurthi, A. Sivasubramaniam, M. J. Irwin, N. Vijaykrishnan and M. Kandemir, "Using complete machine simulation for software power estimation: The SoftWatt approach," in *High-Performance Computer Architecture, 2002. Proceedings. Eighth International Symposium*, 2002.

[18] D. Brooks, V. Tiwari and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Computer Architecture, 2000. Proceedings of the 27th International Symposium*, 2000.

[19] A. Ahmadinia, B. Ahmed and T. Arslan, "A state based framework for efficient system-level power estimation of costum reconfigurable cores.," in *System-on-Chip, 2008. SOC 2008. International Symposium*, 2008.

[20] S. Chandra, K. Lahiri, A. Raghunathan and S. Dey, "Variation-Aware System-Level Power Analysis," in *Very Large Scale Integration (VLSI) Systems, IEEE Transactions*, 2010.

[21] T. Givargis, F. Vahid and J. Henkel, "Instruction-based system-level power evaluation of system-on-a-chip peripheral cores," in *Very Large Scale Integration (VLSI) Systems, IEEE Transactions*, 2002.

[22] J. Chen and L. Thiele, "Task partitioning and platform synthesis for energy efficiency," in *Embedded and Real-Time Computing Systems and Applications, 2009. RTCSA '09. 15th IEEE International Conference*, 2009.

[23] J. Ou and V. K. Prasanna, "Rapid energy estimation of computations on FPGA based soft processors," in *SOC Conference, 2004. Proceedings. IEEE International*, 2004.

[24] V. Tiwari and M. Tien-Chien Lee, "Power analysis of a 32-bit embedded microcontroller," in *Design Automation Conference, 1995. Proceedings of the ASP-DAC '95/CHDL '95/VLSI '95., IFIP International Conference on Hardware Description Languages; IFIP International Conference on very Large Scale Integration., Asian and South Pacific*, 1995.

[25] M. T. Lee, V. Tiwari, S. Malik and M. Fujita, "Power analysis and minimization techniques for embedded DSP software," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions,* vol. 5, pp. 123-135, 1997.

[26] A. Mathur, S. Roy, R. Bhatia, A. Chakraborty, V. Bhargava and J. Bhartia, "JouleQuest: An accurate power model for the StarCore DSP platform," in *VLSI Design, 2007. Held Jointly with 6th International Conference on Embedded Systems., 20th International Conference*, 2007.

[27] S. Roy, R. Bhatia and A. Mathur, "An accurate energy estimation framework for VLIW processor cores," in *Computer Design, 2006. ICCD 2006. International Conference*, 2006.

[28] V. Tiwari, S. Malik and A. Wolfe, "Power analysis of embedded software: A first step towards software power minimization," in *Computer-Aided Design, 1994., IEEE/ACM International Conference*, 1994.

[29] J. Kim, K. Kang, H. Shim, W. Hwangbo and C. Kyung, "Fast estimation of software energy consumption using IPI(inter-prefetch interval) energy model," in *Very Large Scale Integration, 2007. VLSI - SoC 2007. IFIP International Conference*, 2007.

[30] C. Brandolese, S. Corbetta and W. Fornaciari, "Software energy estimation based on statistical characterization of intermediate compilation code," in *Low Power Electronics and Design (ISLPED) 2011 International Symposium*, 2011.

[31] E. Eileen , "UMC Delivers Leading-edge 65nm FPGAs to Xilinx," HSINCHU, Taiwan, 2006.

[32] Xilinx Inc., "ISim User Guide," 2010.

[33] T. Laopoulos, P. Neofotistos, C. A. Kosmatopoulos and S. Nikolaidis, "Measurement of current variations for the estimation of software-related power consumption [embedded processing circuits]," *Instrumentation and Measurement, IEEE Transactions,* vol. 52, pp. 1206-1212, 2003.

[34] P. Sailer, D. R. Kaeli and P. M. Sailer, The DLX Instruction Set Architecture Handbook, 1 ed., San Francisco: Morgan Kaufmann Publishers Inc., 1996.

[35] Xilinx Inc., "MicroBlaze Processor Reference Guide," 2010.

[36] C. Chakrabarti and D. Gaitonde, "Instruction level power model of microcontrollers," in *Circuits and Systems, 1999. ISCAS '99. Proceedings of the 1999 IEEE International Symposium*, 1999.

[37] Y. Hwang, S. Abdi and D. Gajski, "Cycle-approximate Retargetable Performance Estimation at the Transaction Level," in *DATE08*, 2008.