

REDUCING ENERGY CONSUMPTION IN C PROGRAMS THROUGH REGISTER AND
SHARED VARIABLES

Mr. Krisada Samrittianusorn



จุฬาลงกรณ์มหาวิทยาลัย

CHULALONGKORN UNIVERSITY

A Thesis Submitted in Partial Fulfillment of the Requirements

for the Degree of Master of Science Program in Computer Science and Information

Technology

Department of Mathematics and Computer Science

Faculty of Science

Chulalongkorn University

บทคัดย่อและแฟ้มข้อมูลฉบับเต็มของวิทยานิพนธ์ฉบับนี้ได้รับการตีพิมพ์ปี 2554 ที่ให้บริการในคลังปัญญาจุฬาฯ (CUIR)

เป็นแฟ้มข้อมูลของวิทยานิพนธ์ฉบับนี้ที่ส่งผ่านทางบัณฑิตวิทยาลัย

The abstract and full text of theses from the academic year 2011 in Chulalongkorn University Intellectual Repository (CUIR) are the thesis authors' files submitted through the University Graduate School.

การลดความสิ้นเปลืองพลังงานในโปรแกรมภาษาซีด้วยตัวแปรรีจิสเตอร์และตัวแปรร่วม



นายกฤษฎา สัมฤทธิยานุสรณ์

จุฬาลงกรณ์มหาวิทยาลัย

CHULALONGKORN UNIVERSITY

วิทยานิพนธ์นี้เป็นส่วนหนึ่งของการศึกษาตามหลักสูตรปริญญาวิทยาศาสตรมหาบัณฑิต
สาขาวิชาวิทยาการคอมพิวเตอร์และเทคโนโลยีสารสนเทศ ภาควิชาคณิตศาสตร์และวิทยาการ

คอมพิวเตอร์

คณะวิทยาศาสตร์ จุฬาลงกรณ์มหาวิทยาลัย

ปีการศึกษา 2556

ลิขสิทธิ์ของจุฬาลงกรณ์มหาวิทยาลัย

Thesis Title	REDUCING ENERGY CONSUMPTION IN C PROGRAMS THROUGH REGISTER AND SHARED VARIABLES
By	Mr. Krisada Samrittianusorn
Field of Study	Computer Science and Information Technology
Thesis Advisor	Associate Professor Peraphon Sophatsathit, Ph.D.

Accepted by the Faculty of Science, Chulalongkorn University in Partial
Fulfillment of the Requirements for the Master's Degree

.....Dean of the Faculty of Science
(Professor Supot Hannongbua, Dr.rer.nat.)

THESIS COMMITTEE

.....Chairman
(Professor Chidchanok Lursunsap, Ph.D.)

.....Thesis Advisor
(Associate Professor Peraphon Sophatsathit, Ph.D.)

.....External Examiner
(Associate Professor Dumras Wongsawang, Ph.D.)

กฤษฎา สัมฤทธิยานุสรณ์ : การลดความสิ้นเปลืองพลังงานในโปรแกรมภาษาซีด้วยตัวแปรรีจิสเตอร์และตัวแปรร่วม. (REDUCING ENERGY CONSUMPTION IN C PROGRAMS THROUGH REGISTER AND SHARED VARIABLES) อ.ที่ปรึกษาวิทยานิพนธ์หลัก: รศ. ดร. พิระพนธ์ โสภัสสฤติย์, 71 หน้า.

การใช้พลังงานทั่วโลกนั้นเพิ่มขึ้นเรื่อยๆ หนึ่งในสาเหตุที่ก่อให้เกิดปัญหาก็คืออุปกรณ์อิเล็กทรอนิกส์ต่างๆ เช่นคอมพิวเตอร์ส่วนบุคคล อุปกรณ์ฝังตัว อุปกรณ์พกพา และโทรศัพท์มือถือ การลดการใช้พลังงานที่มีประสิทธิภาพเกี่ยวข้องกับฮาร์ดแวร์และซอฟต์แวร์ งานวิจัยนี้พิจารณาการลดพลังงานซอฟต์แวร์โดยเน้นวิธีการเขียนโปรแกรม เทคนิคที่นำเสนอจะอิงตามหลักการจัดสรรตัวแปรเฉพาะที่ในโปรแกรมภาษา C โดยใช้ประโยชน์จากข้อได้เปรียบของหน่วยความจำที่ใช้ร่วมกันและตัวแปรรีจิสเตอร์

การทดลองเป็นการทดสอบโปรแกรมตัวอย่าง 24 โปรแกรม โดยเปรียบเทียบระหว่างการในตัวแปรเฉพาะที่ การใช้ตัวแปรของหน่วยความจำร่วมกัน และการใช้ตัวแปรรีจิสเตอร์

การวิเคราะห์จะดำเนินการในระดับคำสั่งเครื่องเพื่อคำนวณหาปริมาณการบริโภคพลังงาน ผลการวิจัยแสดงให้เห็นว่าหน่วยความจำที่ใช้ร่วมกันเป็นตัวเลือกที่ดีที่สุดโดยลดการจัดสรรที่ซ้ำซ้อนและการเข้าถึงหน่วยความจำ จึงทำให้เกิดการบริโภคพลังงานน้อยกว่าและประมวลผลคำสั่งได้เร็วกว่า

จุฬาลงกรณ์มหาวิทยาลัย

CHULALONGKORN UNIVERSITY

ภาควิชา	คณิตศาสตร์และวิทยาการคอมพิวเตอร์	ลายมือชื่อนิสิต
สาขาวิชา	วิทยาการคอมพิวเตอร์และเทคโนโลยีสารสนเทศ	ลายมือชื่อ อ.ที่ปรึกษาวิทยานิพนธ์หลัก

5373601923 : MAJOR COMPUTER SCIENCE AND INFORMATION TECHNOLOGY

KEYWORDS: ENERGY CONSUMPTION

KRISADA SAMRITTIYANUSORN: REDUCING ENERGY CONSUMPTION IN C PROGRAMS THROUGH REGISTER AND SHARED VARIABLES. ADVISOR: ASSOC. PROF. PERAPHON SOPHATSATHIT, Ph.D., 71 pp.

Energy consumption around the world increases exponentially. One of the causes to blame is electronic devices such as personal computers, embedded devices, and smartphones. To reckon with reducing energy consumption involves efficient hardware and software. This research focuses on the software part, in particular, how to write a program that is energy efficient. The proposed technique is based primarily on local variable reallocation in C programs to exploit the advantages of global variables and register variables. The experiment was conducted on 24 test programs by comparing between local variables and modified program using global variables and register variables.

Analysing the amount of energy consumed is performed at the instruction level. It was found that global variables were the best choice. The benefits are fewer redundant allocations and memory accesses, thereby less energy will be consumed and will help the program execute faster.

จุฬาลงกรณ์มหาวิทยาลัย
CHULALONGKORN UNIVERSITY

Department: Mathematics and Student's Signature

Computer Science Advisor's Signature

Field of Study: Computer Science and
Information Technology

Academic Year: 2013



จุฬาลงกรณ์มหาวิทยาลัย
CHULALONGKORN UNIVERSITY

ACKNOWLEDGEMENTS

I would like to express my uttermost appreciation to my advisor Assoc. Prof. Dr. Peraphon Sophatsathit and to my parents who have given me the opportunity and support to pursue advanced study. I would also like to extend my appreciation to my boss Mr. Ukritv Visitkitjakarn for giving me support to study alongside my work.



CONTENTS

	Page
THAI ABSTRACT	v
ENGLISH ABSTRACT	vii
ACKNOWLEDGEMENTS	viii
CONTENTS	ix
LIST OF FIGURES	1
LIST OF TABLES	1
Chapter 1	1
1.1 The rise of Electrical Energy Demands	1
1.2. Objective	4
1.3. Scope of the work	4
1.4. Expected Outcomes	4
Chapter 2	5
2.1 Related work	5
Chapter 3	9
3.1 Proposed Methodology	9
Chapter 4	23
4.1 Experimental procedure	23
4.2 First case study (1) - Function call	25
4.3 Second case study (2) - Repeated Function calls	32
4.4 Third case study (3) - Function calls Function	34
4.5 Fourth case study (4) – Nested repeated Function calls	37
4.6 Discussion	48
Chapter 5	55

	Page
5.1 Conclusion and future work.....	55
REFERENCES	56
Appendix A.....	58
Appendix B.....	66
VITA.....	72



จุฬาลงกรณ์มหาวิทยาลัย
CHULALONGKORN UNIVERSITY

LIST OF FIGURES

	Page
Figure 1.1 Annual energy demand by region.....	1
Figure 1.2 Energy needed and availability.....	2
Figure 1.3 Global installed base of Desktop PCs + Notebooks PCs vs. Smartphones + Tablets, 2009 – 2015E.....	3
Figure 3.1 Find total of instruction.....	9
Figure 3.2 A comparative energy consumed by local variable, shared variable and register variable.....	11
Figure 3.3 Case 1 Pseudocode - function calls.....	12
Figure 3.4 Example of instruction in case (1).....	13
Figure 3.5 Case (2) Pseudocode - Repeated function calls.....	14
Figure 3.6 Example of instruction in case (2).....	15
Figure 3.7 Case (3) Pseudocode - function calls to other functions.....	16
Figure 3.8 Example of instruction in case (3).....	17
Figure 3.9 Case (4) Pseudocode - Nested repeated function calls to another function	18
Figure 3.10 Example of instruction in case (4).....	19
Figure 3.11 Simple C source code using local variable.....	20
Figure 4.1 KP tool functionality.....	24
Figure 4.2. Case 1 - C source code using local variable.....	25
Figure 4.3 KP running case study 1 - local variable.....	26
Figure 4.4 case study 1 - assembly instruction as shown in the KP tool.....	27
Figure 4.5 Case study 1 - C source code using shared variable.....	28
Figure 4.6 KP running case study 1 – Shared variable.....	29
Figure 4.7 Case study 1 - C source code using register variables.....	30
Figure 4.8 KP running case study 1 – register variable.....	31
Figure 4.9 KP running case study 2 – local variable.....	32

Figure 4.10 KP running case study 2 – shared variable	33
Figure 4.11 KP running case study 2 – register variable	34
Figure 4.12 KP running case study 3 – local variable.....	35
Figure 4.13 KP running case study 3 – shared variable	36
Figure 4.14 KP running case study 3 – register variable	37
Figure 4.15 KP running case study 4 – local variable.....	38
Figure 4.16 KP running case study 4 – shared variable	39
Figure 4.17 KP running case study 4 – register variable	40
Figure 4.18 Comparison of number of instruction (Y) for all cases (X).....	44
Figure 4.19 Comparison of energy consumption (Y) for all cases (X).....	44
Figure 4.20 Comparison of clock cycle (RT) (Y) for all cases (X).....	45
Figure 4.21 Comparison of clock cycle (L) (Y) for all cases (X).....	45
Figure 4.22 Comparison of clock (RT + L) (Y) for all cases (X)	46
Figure 4.23 corresponds to Table 4.6	47
Figure 4.24 Comparison of number of instruction (Y) for all cases (X)	51
Figure 4.25 Comparison of energy consumption (Y) for all cases (X).....	51
Figure 4.26 Comparison of clock cycle (RT) for all cases (X)	52
Figure 4.27 Comparison of clock cycle (L) (Y) for all cases (X).....	52
Figure 4.28 Comparison of clock cycle (L+RT) (Y) for all cases (X).....	53

LIST OF TABLES

	Page
Table 3.1 Clock cycles for an instruction.....	10
Table 3.2 Assembly instruction and number of count.....	21
Table 4.1 Number of instructions.....	41
Table 4.2 Energy consumption (nJ) by allocation schemes.....	41
Table 4.3 Number of clock cycle (RT) by allocation schemes.....	41
Table 4.4 Number of clock cycle (L) by allocation schemes.....	42
Table 4.5 Number of clock cycle (RT + L) by allocation schemes.....	42
Table 4.6 Showing all results from 4 cases.....	47
Table 4.7 Number of instructions (set 2).....	49
Table 4.8 Energy consumption by allocation scheme (set 2).....	49
Table 4.9 (RT) Number of clock cycle by allocation scheme (set 2).....	50
Table 4.10 (L) Number of Clock cycle by allocation scheme (set 2).....	50
Table 4.11 (RT + L) Number of clock cycle (set 2).....	50

Chapter 1

Introduction

1.1 The rise of Electrical Energy Demands

High population growth over the years and the emergence of large economies such as China and India have led to higher demand of natural resources and have caused the world's energy consumption to skyrocket. Private and public sectors around the world are committed to renewable energy sources to cope with the increasing demand. In the transportation sector, many traditional technologies are progressively being shifted toward using cleaner energy with less pollution. This includes the shift from fossil fuel to electrical and hybrid energy [1].

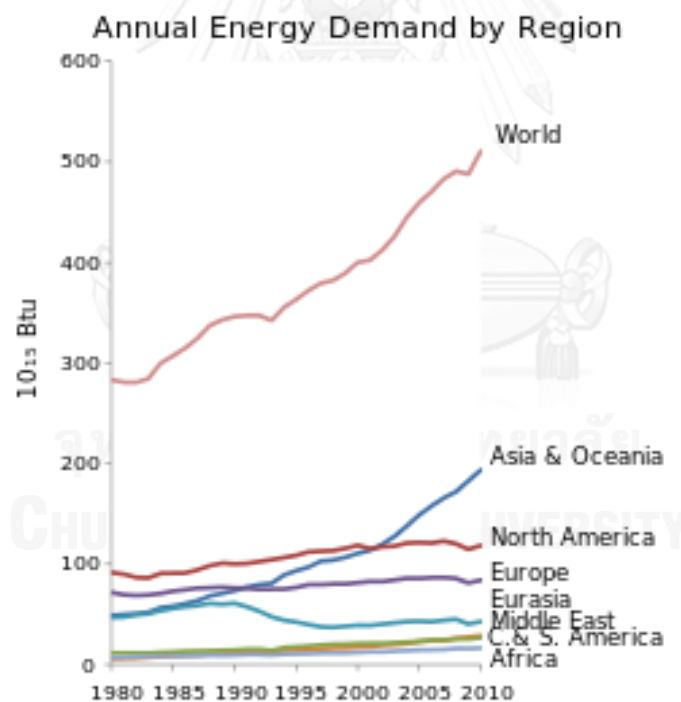


Figure 1.1 Annual energy demand by region

Source: <http://static2.businessinsider.com/image/50bd70cc.jpg>

The arrival of smartphones and various portable devices have raised electrical energy consumption considerably. As the devices have improved in both speed and

computational power and features, demands for energy consumption of these devices escalate at a rapid rate. Unfortunately, the battery technology still cannot cope with the energy consumption, especially in a commercial environment. This trend has become more apparent as external battery supplies are needed. The gap of energy needed and its availability is exemplified in Figure 1.2

Moreover, a shift from traditional printed media to electronic media is prominent and widely adopted now in many countries. The larger the screen size and the faster the CPU mean higher energy consumption and demand. Other Internet-connected and wearable devices have followed suit. Therefore, methods to minimize current power drain are called for [2].

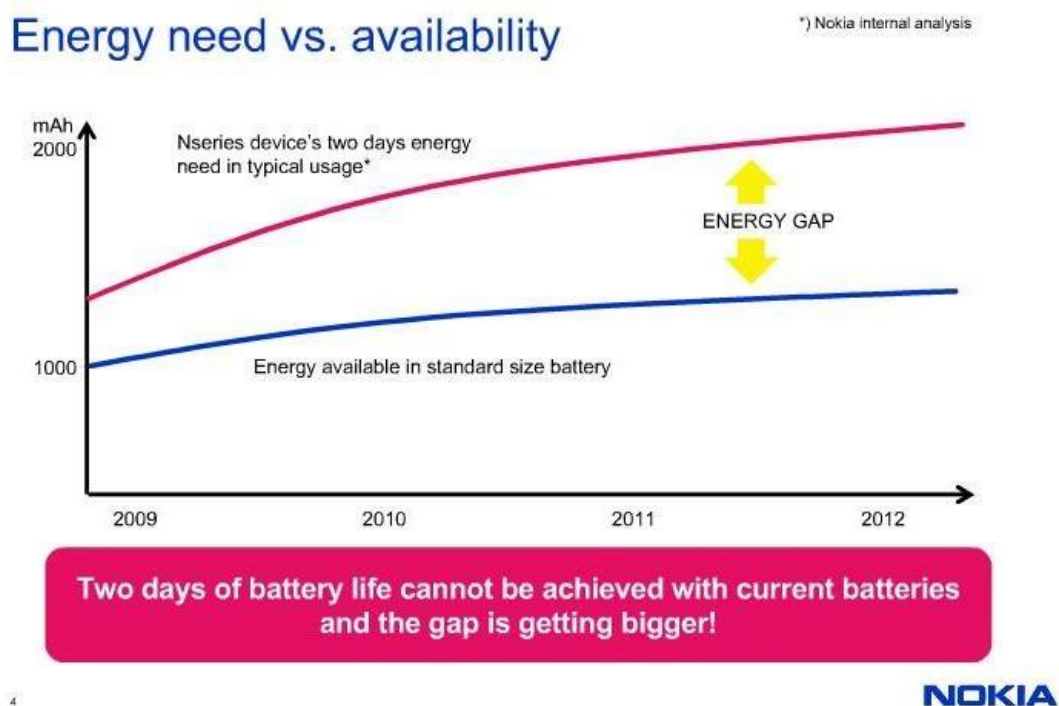


Figure 1.2 Energy needed and availability

Source: <http://static2.businessinsider.com/image/1200.jpg>

Figure 1.3 shows the significant shift from desktop computers connected to a power source to smart mobile/portable devices around the world. This means the issue of energy efficiency is much more important than before as the number of

portable device users grows very rapidly and has definitely signified toward the next emerging technology [3].

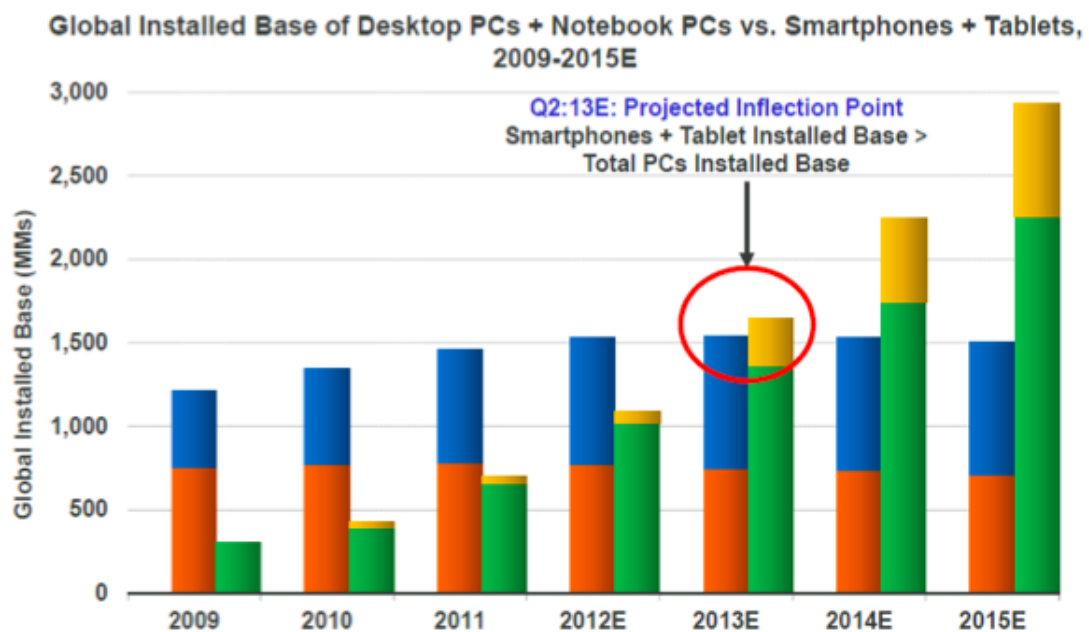


Figure 1.3 Global installed base of Desktop PCs + Notebooks PCs vs. Smartphones + Tablets, 2009 – 2015E

Source: <http://static2.businessinsider.com/image/c000005.jpg>

As a consequence, many researches are underway to find ways to save the energy needed by electronic devices. Broadly speaking, the research is divided into 2 groups, namely, hardware and software. Studies have shown that software is the principal factor of energy consumption in computer systems [4].

A typical computer program in execution stores, retrieves, and processes variables such as local variables, shared variables, and register variables. Heavy use of these variables wastes considerable energy. One remedy is code modification to reorganize of original code to properly allocate variables and parameters, thereby balancing the distribution of energy consumption.

This research is specifically targeted on how the software can be designed and architected with energy consumption as part of the design. C Programming Language, a widely used and very portable programming language, is used in this investigation.

1.2. Objective

This study focuses on the reduction of energy consumed by computer programs by applying code modification to shared variables and register variables.

1.3. Scope of the work

This research will confine the scope within the following constraints:

1. Limit to C programming language.
2. Focus on local stack, register variables, and shared variables.
3. Use is Intel® Core 2 Duo system running Windows 7 as the working environment
4. The unit of measure is instruction clock cycle.

1.4. Expected Outcomes

The proposed technique will offer the following benefits

1. Reduce energy consumption by computer programs,
2. Improve program performance precipitating from (1), and
3. Compact program/code organization.

Chapter 2

Related work

2.1 Related work

Energy consumption is one of the critical factors for modern portable device designs. Often, one of the key performance indicators widely used in the industries is energy consumption, i.e. the current drain. Researches and studies from the academic sectors and the industrial sectors have been focusing on improving measurements of the energy consumption in hardware or software or a combination of both despite the complexity of the systems. Some research papers discuss source code analysis and coding techniques while others focus on better software architecture design. This research also explores different measurement tools.

Sheayun Lee, Adreads Ermedahl, et al [4] showed a technique for finding an accurate energy consumption model at the instruction level using combined statistical analysis technique and empirical method to estimate the energy consumption of an instruction. However, it was necessary to analyze the characteristics of memory devices since the energy consumption was also dependent on it.

Optimization of software solutions called POWERAPI, estimates the power consumption of processes and applications according to different dimensions (CPU, network, etc.). Adel Nouredine and Aurelien Bourdon [5] used this library to study the impact of programming languages and algorithmic choices on energy consumption. However, they needed to propose more energy models for other hardware resources (such as memory and disk) and used power-aware information to adapt application at runtime based on energy concerns.

PowerScope is an energy profiling tool which was proposed by Jason Flinn and M. Satyanarayanan. PowerScope [6] profiles CPU cycles of specific process and procedures in software. The approach utilizes hardware instrumentation to measure current levels with kernel software support to perform statistical sampling of system activity. It is able to pinpoint the key energy consumption source and hence reduce the energy consumption of an adaptive video playing application. However, it needs

further exploration and better model of the relationship between energy usage and battery life.

Nadine and Bill [7] proposed Green Tracker, a tool for estimating the energy consumption of installed software systems. Green Tracker utilizes a benchmarking test to determine which software systems are the most efficient given the user's current computer configuration.

Thanh Do, Suhil Rawshdeh, et al [8] proposed a tool called PTOPI which was a process-level power profiling tool. The tool provides the power consumption (in Joules) of the running processes. For each process, it gives the power consumption of the CPU, network interface, computer memory, and hard disk. The tool consists of a daemon running in kernel space and continuously profiling resource utilization of each process. For the CPU, it also uses TDP provided by constructors in the energy consumption calculations. It then calculates the amount of energy consumed by each application for a period of time.

Various techniques have been attempted to cope with measuring power consumption at instruction level problem. A simple yet effective technique will help extend the battery life on mobile devices by controlling data access. Eugene Shih, Paramvir Bahl, et al [9] introduced a method to extend the battery lifetime by reducing its idle power, the power a device consumed in a standby state. To reduce this, the wireless network card was shut down when it was not being used.

From a software standpoint, proper management of memory allocation and access will help reduce the amount of energy consumption [10]. It involves the problem of allocating memory to variables in embedded Digital Signal Processing software to maximize data transfers from different memory banks to registers.

Mike Tien-Chien, L. and V. Tiwaris [10] showed a software analysis tool that had a method to compile program into the instruction level and analyzed it at instruction level.

David Binkley [11] showed trends of source code analysis to extract information from the source code to help a programmer analyze their program's performance and tweak it. There were choices to tweak from changing high level source, recompiling,

re-tweaking, or performing the change on the lower-level assembly code or abandoning the tweaking.

Since C programming language was developed in 1972, the language has been widely used and very portable for the majority of hardware platforms today. It is a popular language of choice to implement software in embedded devices compared to ASM. C is a very unique high-level language that still provides low level control especially on memory utilization and can generate a compact-size executable which is suitable for small memory footprint devices.

As technology today has come to hand held portable devices, application development uses high-level programming language such as Java or Objective C. Programmer can still use native code like C because it is better for computationally intensive algorithms such as game development and visual computing [12, 13].

Tim A. Wagner, Vance Maverick, et al [14] conducted research using C language as a primary tool for analysing each function in machine language that the GNU C Compiler generated.

John Max Skaller [15] discussed the introduction of nested functions into C/C++. Nested functions were well understood and their introduction required little effort from either compiler vendors or programmers. Nested functions offer significant advantages, including rapid prototyping and functional decomposition, as well as gains in both processor and programmer performance.

Yanbing Li and Henkel J. [16] showed combinations and sequences of transformations that yielded the most energy savings under memory size constraints, evaluated the impact of transformations, and estimated the energy used by code segment that contained repeated loop and procedure calls.

Tiwari V., Malik S., et al [17] mentioned that power constraints were increasingly becoming the critical component of computer design specifications. They described a framework for energy estimation of a program using the instruction level power model. They showed the average current and the number of cycles for each to determine the power used.

Some research efforts analyse basic aspects of the way programs manipulate the runtime stack. Cullen Linn, Saumya Debray, et al [18], and Thomas Reps and Gogul

Balakrishnan [19] showed how the runtime stack was used and the stack behaviour of a function. For each function call in C, they showed how stack locations stored values of functions and parameters.

Peter Sestoft [20] investigated when function parameters can be safely replaced by global variables. It showed the benefits of using global variables to reduce the time and space cost of stack allocation of function parameters whenever possible. As function parameters are replaced by global variables, using the stack is more expensive in terms of run-time and storage consumption than fixed global allocation.

Jack W. Davidson and David B. Whalley [21] also mentioned that when using registers to store variables, the number of instructions executed was affected by two factors. Typically, as more variables were allocated to registers, the number of instructions used for saving and restoring registers increased. On the other hand, as frequently used variables were allocated to registers, the number of instructions aside from those used for saving and restoring registers decreased.

At a finer grained level, Grochowki and Annavaram [22] analyzed energy per instruction (EPI) based on an Intel processor. They described Energy per Instruction (EPI) as a measure of the amount of energy expended by a microprocessor for each instruction that the microprocessor executed. They explained the factors that affected a microprocessor's EPI and derived a historical comparison of the trends in EPI over multiple generation of Intel processors.

Chapter 3

Proposed Methodology

3.1 Proposed Methodology

To explore the operating characteristics of parameter and variable allocation at the instruction level, a C program is first compiled into assembly code by using a compiler to inspect the clock cycle and the number of instructions. The total number of instructions used by the entire program can then be converted to energy consumption, which is measured as energy per instruction (EPI). The unit of EPI is expressed in Joules.

According to [22], the value of EPI of selected Intel CPU were investigated. The total energy consumption of the program can be calculated using the formula below.

$$\text{Total energy consumption} = \text{EPI} \times \text{Total number of instructions}$$

Figure 3.1 shows a calculation example of the total number of instructions in pseudo code of assembly language. The left column shows instruction set which is compiled from a C program. Then a simple count is made from first to last instructions. The total number of instructions is $n + m$.

<i>Label 1:</i>	Count
Instruction A	1
Instruction B	2
...	...
Instruction C	n
<i>Label 2:</i>	
Instruction D	1
Instruction E	2
...	...
Instruction F	m
Total number of Instructions	n + m

Figure 3.1 Find total of instruction

This study uses the total clock cycle for estimating the speed of the program. Each machine instruction is examined to determine the number of clock cycles. Each instruction has different numbers of clock cycles depending on the type of instruction as shown in the example in Table 3.2. Reciprocal throughput (RT) is the average number of core clock cycles per instruction for a series of independent instructions of the same kind in the same thread [23]. Latency (L) of an instruction is the delay that the instruction generates in a dependency chain. The measurement unit is clock cycles [23]. The total clock cycles of every instruction can be summed for measuring instruction speed.

Table 3.1 Clock cycles for an instruction

Instruction	Clock cycle (RT)	Clock cycle (L)
MOV r,m	1	2
PUSH r	1	3
POP m	1.5	2
INC	0.33	1
ADD r,r	0.33	1

This research will focus on finding energy consumption of allocating variable in local, shared variable, and register in C programming. The amount of energy consumed by local variable allocation is analyzed by comparing shared variable and register memory utilization. For this study, a set of programs that perform the same functionality are written in three different ways. Each program uses either local, global, or register variables. Figure 3.1 illustrates such an arrangement.

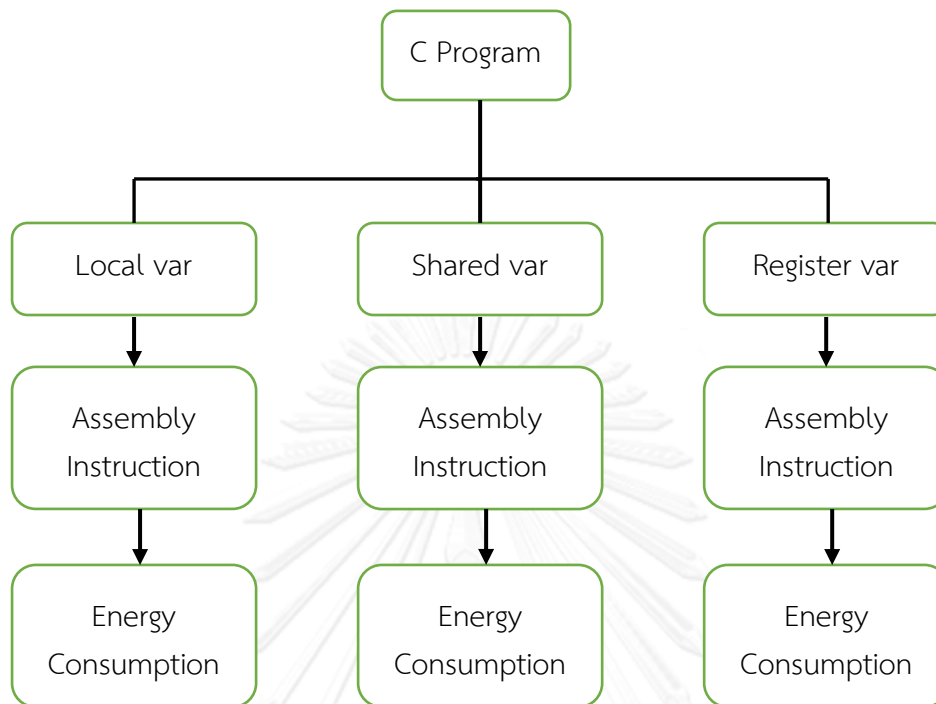


Figure 3.2 A comparative energy consumed by local variable, shared variable and register variable

The above study demonstrates the impact of memory utilization to the energy consumption as well as the speed of program execution.

There are two scenarios to be investigated, namely, (1) local variable vs. shared variable, and (2) local variable vs. register variable. A collection of C programs are set up to assist in the analysis. The following case studies will be carried out to exercise both scenarios. Each case study will show the energy consumption and the total clock cycle used. There are 4 cases of programming function deployed in each scenario:

1. Function calls
2. Repeated function calls
3. Function calls to function
4. Repeated function calls to function

The explanation of these 4 cases is described in details.

1. *Function calls*. This is the simplest exercise for parameter allocation, access, and retrieval. Normally, a programmer will use local variables declared in the main function. These variables will subsequently be passed to other functions in the form of parameters. In scenario (1), program modification is done by moving local variables to shared variables, thereby no parameter passing is needed. In scenario (2), the register keyword is simply added to proper local variables.

In the ‘Shared variable’ column of Figure 3.3, the variable declaration is moved out of the main scope to shared variable or global scope. Then the parameters of the function can be removed.

Local	Shared variable	Register
Declare function (type,..)	Declare function (type,..) Declare global variable	Declare function (type,..)
Main function Declare local variable Statement Call function (argument A,..) End main Function (parameter A,..){ }	Main function Statement Call function () End main Function (){ }	Main function Declare register local variable Statement Call function (argument A,..) End main Function (parameter A,..){ }

Figure 3.3 Case 1 Pseudocode - function calls

The Figure 3.4 shows a simplified version of the instruction set and its corresponding executing cycle. The number of instructions and the clock cycles consumed in executing the program is used to illustrate how energy consumption is calculated. In this case, finding the total number of instructions can be obtained from

Total number of instructions = $n + m$

The energy consumption in Joules can be calculated as:

$$\text{Energy consumption} = (n + m) \times \text{EPI}$$

Total clock cycle of RT and become

$$\text{Total clock cycle (RT)} = x + y$$

$$\text{and (L)} = a + b$$

<i>main:</i>	Count	Clock (RT)	Clock (L)
Instruction A	1	1	1
Instruction B	2	2	2
...
Instruction C	n	x	a
<i>Label:</i>			
Instruction D	1	1	1
Instruction E	2	2	2
...
Instruction F	m	y	b

Figure 3.4 Example of instruction in case (1)

2. *Repeated function calls.* The objective is to find code segments that exhibit high energy consumption in a program and the behaviour of the associated variables/parameters. As such, program improvement can be directed to the right area where energy consumption can be reduced. This case intentionally contrives repeated calls to function for this particular purpose.

In Figure 3.5 the pseudocode shows a loop containing a function inside within the main function. This will call a function up to n times.

Local	Shared variable	Register
Declare function (type,..)	Declare function (type,..)	Declare function (type,..)
Main	Declare global variable	Main
Declare local variable	Main	Declare register local
Statement	Statement	variable
For i to n	For i to n	Statement
Call function(argument A,..)	Call function()	For i to n
End for	End for	Call function(argument A,..)
End main	End main	End for
Function (parameter A,..){	Function (){	End main
}	}	Function (parameter A,..){
		}

Figure 3.5 Case (2) Pseudocode - Repeated function calls

A similar case study can be seen in Figure 3.6. However, this case considers the loop depending on how the instruction jump to the next label as shown below.

In this case, the total number of instructions is:

$$\text{Number of total instruction} = n + m + 1 (k)^i$$

where i is the number of repetitions depending on the conditions.

The energy consumption in Joule can be calculated as

$$\text{Energy consumption} = n + m + 1 (k)^i \times \text{EPI}$$

Total clock cycles of RT and L become

$$\text{Total clock cycle (RT)} = x + y + 1 (z)*i$$

$$\text{And (L)} = a + b + 1 (c)*i$$

<i>Main:</i>	Count	Clock (RT)	Clock (L)
Instruction A	1	1	1
Instruction B	2	2	2
...
Instruction C	n	x	a
<i>Label:</i>			
Instruction D	1	1	1
Instruction E	2	2	2
...
Instruction F	m	y	b
<i>Label:</i>			
Instruction CMP	1	1	1
<i>Instruction JMP Label</i>	2	2	2
...
<i>Instruction G</i>	k	z	c

Figure 3.6 Example of instruction in case (2)

3. *Function calls to function.* This case is intended to investigate the cascading effect of energy consumption consumed by parameter allocation and reference. The complication of such operations, i.e., stack, shared variable, and register variable, at the instruction level are systematically measured and compared.

Figure 3.7 shows the pseudocode of function calls to other functions. This scenario might occur if a program has many subroutines or functions. In this simple case, function 1 calls function 2.

Local	Shared variable	Register
Declare function (type,..) Main function Declare local variable Statement Call function1 (argument A,...) End main Function1 (parameter A,...){ Call function2 (argument A) } Function2 (parameter B,...){ }	Declare function (type,..) Declare global variable Main function Statement Call function1() End main Function1 (){ Call function2 () } Function2 (){ }	Declare function (type,..) Main function Declare register local variable Statement Call function1(argument A,...) End main Function1 (parameter A,...){ Call function2 (argument A) } Function2 (parameter B,...){ }

Figure 3.7 Case (3) Pseudocode - function calls to other functions

Figure 3.8 shows a similar case study. There are more functions to be called which include function labels.

In this case, finding total number of instructions is

$$\text{Number of total instruction} = n + m + k$$

The energy consumption in Joule can be calculated as

$$\text{Energy consumption} = (n + m + k) \times \text{EPI}$$

Total clock cycles of RT and L become

$$\text{Total clock cycle (RT)} = x + y + z$$

$$\text{And (L)} = a + b + c$$

<i>Main:</i>	Count	Clock (RT)	Clock (L)
Instruction A	1	1	1
Instruction B	2	2	2
...
Instruction C	n	x	a
<i>Label: function</i>			
Instruction D	1	1	1
Instruction E	2	2	2
...
Instruction F	m	y	b
<i>Label: function</i>			
Instruction G	1	1	1
Instruction H	2	2	2
...
Instruction I	k	z	c

Figure 3.8 Example of instruction in case (3)

4. *Nested repeated function calls to function.* This case culminates all of the above complications to demonstrate as close to actual operation as possible.

Figure 3.9 shows pseudocode that contains the loop calls containing a function call to another function.

Local	Shared variable	Register
Declare function (type,..) main function() Declare local variable Statement For i to n Call function1 (argument A,...) End for End main Function1 (parameter A,...){ Call function2 (argument B,...) } Function2 (parameter B,...){ }	Declare function (type,..) Declare global variable main function() Statement For i to n Call function1() End for End main Function1 (){ Call function2 () } Function2 (){ }	Declare function (type,..) main function() Declare register local variable Statement For i to n Call function1(argument A,...) End for End main Function1 (parameter A,...){ Call function2 (argument B,...) } Function2 (parameter B,...){ }

Figure 3.9 Case (4) Pseudocode - Nested repeated function

calls to another function

Figure 3.10 shows a similar case study. This case will consider the number of repetitions (i) depending on how the instruction jump to the next label as below.

In this case, the total number of instructions is:

$$\text{Number of total instruction} = n + k + 1 + (p + m) * i$$

where i is the number of repetitions depending on the condition.

The energy consumption in Joule can be calculated as:

$$\text{Energy consumption} = n + k + 1 + (p + m) * i \times \text{EPI}$$

Total clock cycle of RT and L become

$$\text{The total clock cycle (RT)} = x + z + 1 + (v + y) * i$$

$$\text{And (L)} = a + c + 1 + (d + b) * i$$

<i>Main:</i>	Count	Clock (RT)	Clock (L)
Instruction A	1	1	1
Instruction B	2	2	2
...
Instruction C	n	x	a
<i>Label:</i>			
Instruction D	1	1	1
Instruction E	2	2	2
...
Instruction F	m	y	b
<i>Label:</i>			
Instruction CMP	1	1	1
<i>Instruction JMP Label</i>	2	2	2
...
<i>Instruction G</i>	k	z	c
<i>Label:</i>			
Instruction H	1	1	1
Instruction I	2	2	2
...
Instruction J	p	v	d

Figure 3.10 Example of instruction in case (4)

Figure 3.11 shows sample C code to demonstrate case study 1. This code will be analyzed to determine the energy consumption and clock speed. In this program, variable *m* is declared as a local variable being passed by a function as a parameter.

```

#include <stdio.h>
int function (int);
int main( ) {
int m,r;
m = 10;
r = func (m) ;
return 0;
}
int func (int m) {
return m+1;
}

```

Figure 3.11 Simple C source code using local variable

The C source code is compiled into assembly instructions and is shown in Table 3.3. The first column lists the instruction set. There is a main function label “_main:” and function label “_func:” which contains the instructions used inside the function. To calculate the total number of instructions, just simply count instructions in “_main:” and “_func:” from first to last instruction. There are 13 and 5 instructions in “_main” and “_func” functions, respectively. The total number of instructions is 18. In this research, the experiment used Intel CPU and the EPI value was set to 11 nJ according to [22] and The energy consumption becomes $18 \times 11 = 191$ nJ.

The last 2 columns “Clock (RT)” and “Clock (L)” represent Reciprocal Throughput and Latency. The clock value used by each instruction can be taken from reference data sheets [23]. For example, the `mov m,i` instruction takes 1 RT clock cycle and L with 3 clock cycles. In the `_main:` function, clock (RT) is 11.99 and L is 18. In “_func:”

clock (RT) is 4.33 and L is 6. The total of clock cycles (RT) and L are 16.65 and 24, respectively.

Table 3.2 Assembly instruction and number of count

Instruction set		Instruction	Clock (RT)	Clock (L)
<u>_main:</u>				
push	ebp	1	1	3
mov	ebp, esp	2	0.33	1
and	esp, -16	3	0.33	1
sub	esp, 32	4	0.33	1
call	__main	5	2	
mov	DWORD PTR [esp+28], 10	6	1	3
mov	eax, DWORD PTR [esp+28]	7	1	2
mov	DWORD PTR [esp], eax	8	1	3
call	_func	9	2	
mov	DWORD PTR [esp+24], eax	10	1	3
mov	eax, 0	11	0.33	1
leave		12	1	
ret		13	1	
<u>_func:</u>				
push	ebp	1	1	3
mov	ebp, esp	2	0.33	1
mov	eax, DWORD PTR [ebp+8]	3	1	2
pop	ebp	4	1	
ret		5	1	
Total		18	16.65	24
Total Energy Consumption		191 nJ		

Other examples on shared variable and register variable for memory access are carried out at in a similar manner. The next section will describe an experimental and its results.



Chapter 4

Experimental Results and Discussions

4.1 Experimental procedure

A tool called KP program was built to help analyze the test programs. The tool first reads an input C program submitted by the user under predefined test scenarios. It locates local variables and function parameters and prompts the user to reallocate or alter them to shared variables or register variables as mandated by the test scenarios. A lookup table is created by the tool to hold all the data selected by the user for reallocation or alteration, whichever applies. The tool then compiles both the original and the modified C programs to produce assembly output. In so doing, all instructions are available for determining clock cycles and the EPI equivalent.

Figure 4.1 depicts the design of KP tool. User interface includes: (1) a menu bar for various functionalities such as reset, compile (2) a browse button for selecting an input C program file, and (3) an analyze button for analyzing the input program. The left panel (4) of the tool shows the C program and the right panel (5) shows the assembly instructions. Results of the analysis are shown at the bottom of the right panel (6). Use of this tool will be described in the next section.

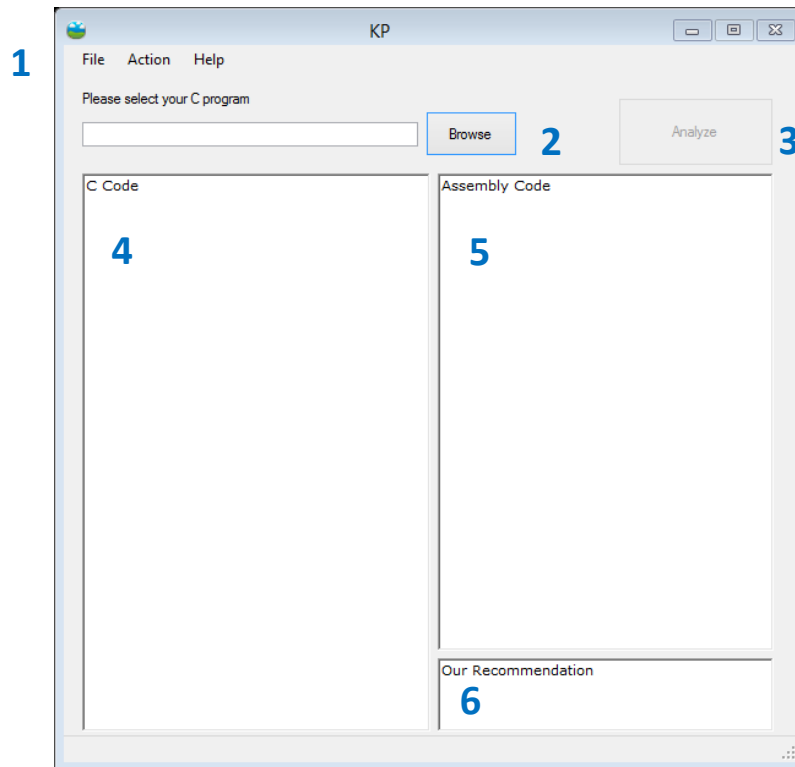


Figure 4.1 KP tool functionality

To run test programs, the operating environment set up for the experiment was hosted by a laptop computer with Intel Core 2 Duo @ 2.00GHz 65 nm, 3 GB RAM running Windows 7. The tool was coded in C# using Microsoft Visual Studio version 2010. All test programs were compiled into Assembly instructions with MinGW [24] which is a ported GNU Compiler collections (GCC). All assembly instructions were based on Intel.

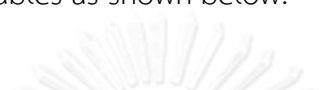
The experiment analyzed 24 sample programs which were divided into 2 sets. In the first set, there were 12 programs with 3 programs for each of the four cases outlined in the previous chapter. For the second set, there were also 12 programs containing more complex code that included more function parameters.

The rationale behind each case study is to determine the amount of energy consumed by program instructions under different functions and to analyse the advantages of using each memory access type (local, shared variable, and register).

The experiment began with simple function calls as seen in case study (1). The tool analyzed and compared clock cycles used, and the energy consumed in each scenario. Code analyses are shown below.

4.2 First case study (1) - **Function call**

The C code uses local variables as shown below.



```
#include <stdio.h>
int func1(int, int, int );
int main(void){
int xint,yint,zint,res1;
xint = 5;
yint = 10;
zint = 15;
res1 = func1(xint, yint, zint);
return 0;
}
int func1(int x, int y, int z){
int temp1;
temp1 = x + y + z;
return temp1;
}
```

Figure 4.2. Case 1 - C source code using local variable

The outputs obtained from KP tool are depicted in Figure 4.3

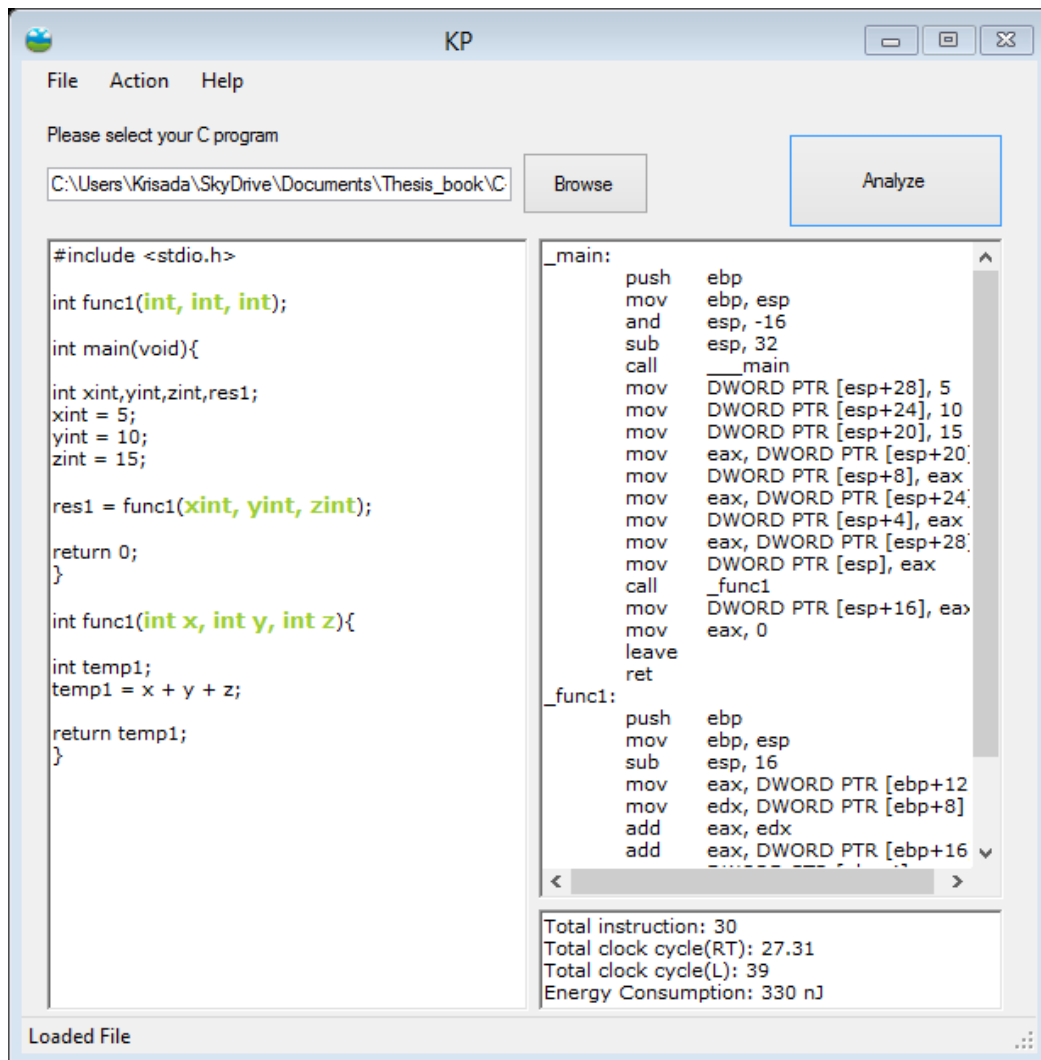


Figure 4.3 KP running case study 1 - local variable

The C program in this figure contained 30 instructions that utilized 27.31 clock cycles (RT) and 39 clock cycles (L). Energy consumption was approximately 330 nJ. In Figure 4.4 shows the entire instruction set which is compiled from C program.

```

_main:
    push    ebp
    mov     ebp, esp
    and     esp, -16
    sub     esp, 32
    call    ___main
    mov     DWORD PTR [esp+28], 5
    mov     DWORD PTR [esp+24], 10
    mov     DWORD PTR [esp+20], 15
    mov     eax, DWORD PTR [esp+20]
    mov     DWORD PTR [esp+8], eax
    mov     eax, DWORD PTR [esp+24]
    mov     DWORD PTR [esp+4], eax
    mov     eax, DWORD PTR [esp+28]
    mov     DWORD PTR [esp], eax
    call    _func1
    mov     DWORD PTR [esp+16], eax
    mov     eax, 0
    leave
    ret

_func1:
    push    ebp
    mov     ebp, esp
    sub     esp, 16
    mov     eax, DWORD PTR [ebp+12]
    mov     edx, DWORD PTR [ebp+8]
    add     eax, edx
    add     eax, DWORD PTR [ebp+16]
    mov     DWORD PTR [ebp-4], eax
    mov     eax, DWORD PTR [ebp-4]
    leave
    ret

```

Figure 4.4 case study 1 - assembly instruction as shown in the KP tool

From the experiment whose results were shown in Figure 4.3, the KP tool highlights the C code in the left panel in green colour when the parameters should be removed.

Figure 4.5 shows the result of this code change where local variables are moved out of the main function. In other words, these variables now become shared variables.

```
#include <stdio.h>
int xint,yint,zint,res1;
void func1(void);
int main(void){
xint = 5;
yint = 10;
zint = 15;
func1();
return 0;
}
void func1(void){
res1 = xint + yint + zint;
}
```

Figure 4.5 Case study 1 - C source code using shared variable

The KP tool in Figure 4.6 shows the analysis of the modified C program using shared variable.

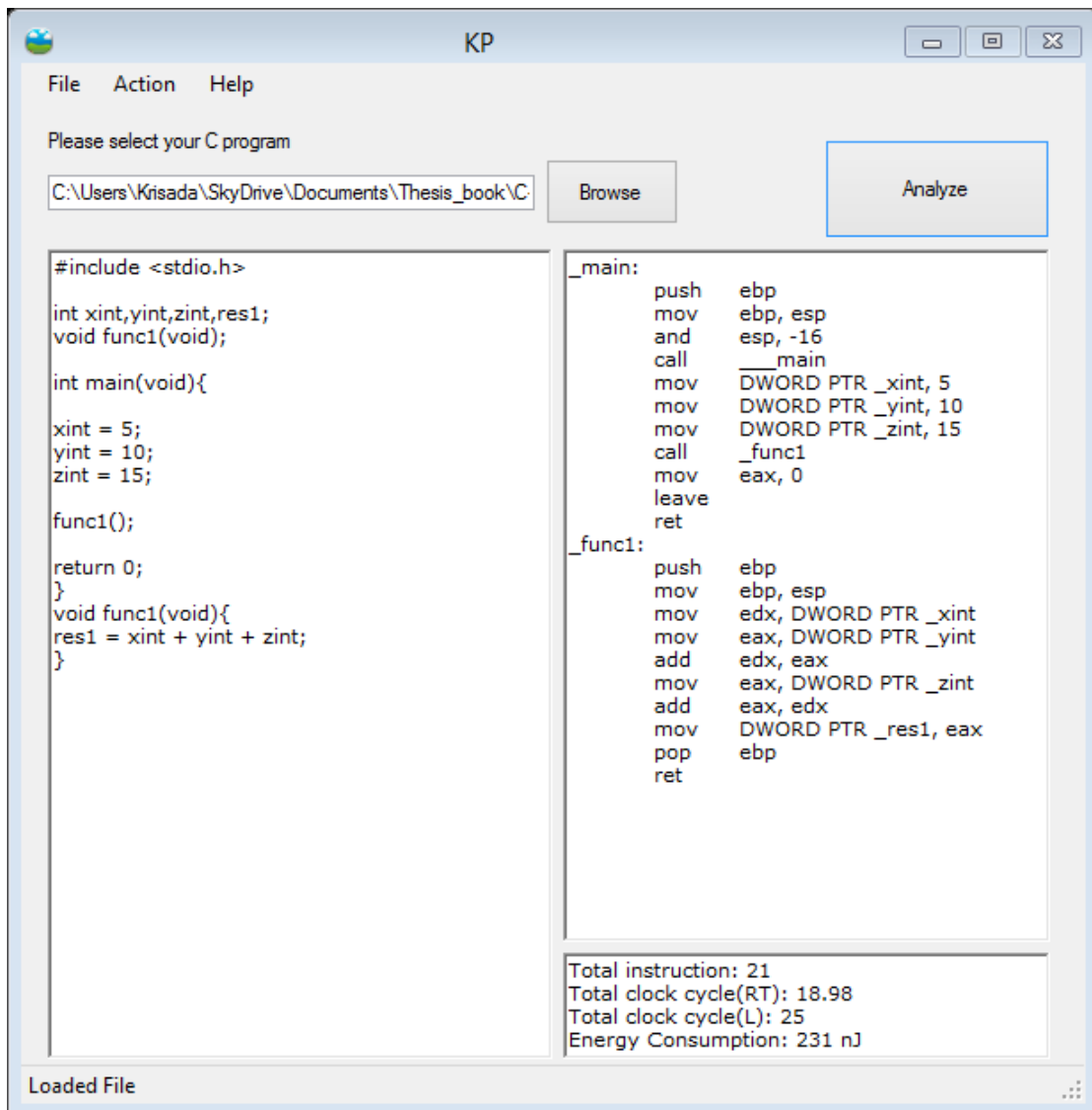


Figure 4.6 KP running case study 1 – Shared variable

The program in Figure 4.6 contains 21 instructions that utilize 18.98 clock cycles (RT) and 25 clock cycles (L) and the energy consumption is about 231 nJ of energy.

Next, the experiment followed scenario 2 where local variables were added using the keyword register. All these variables were stored in the register as shown in Figure 4.7

```
#include <stdio.h>
int func1(int , int , int );
int main(void){
register int xint,yint,zint;
int res1;
xint = 5;
yint = 10;
zint = 15;
res1 = func1(xint, yint, zint);
return 0;
}
int func1(int x, int y, int z){
register int temp1;
temp1 = x + y + z;
return temp1;
}
```

Figure 4.7 Case study 1 - C source code using register

The KP tool in Figure 4.8 shows similar analysis of the previous experiment for the modified C program using register variables.

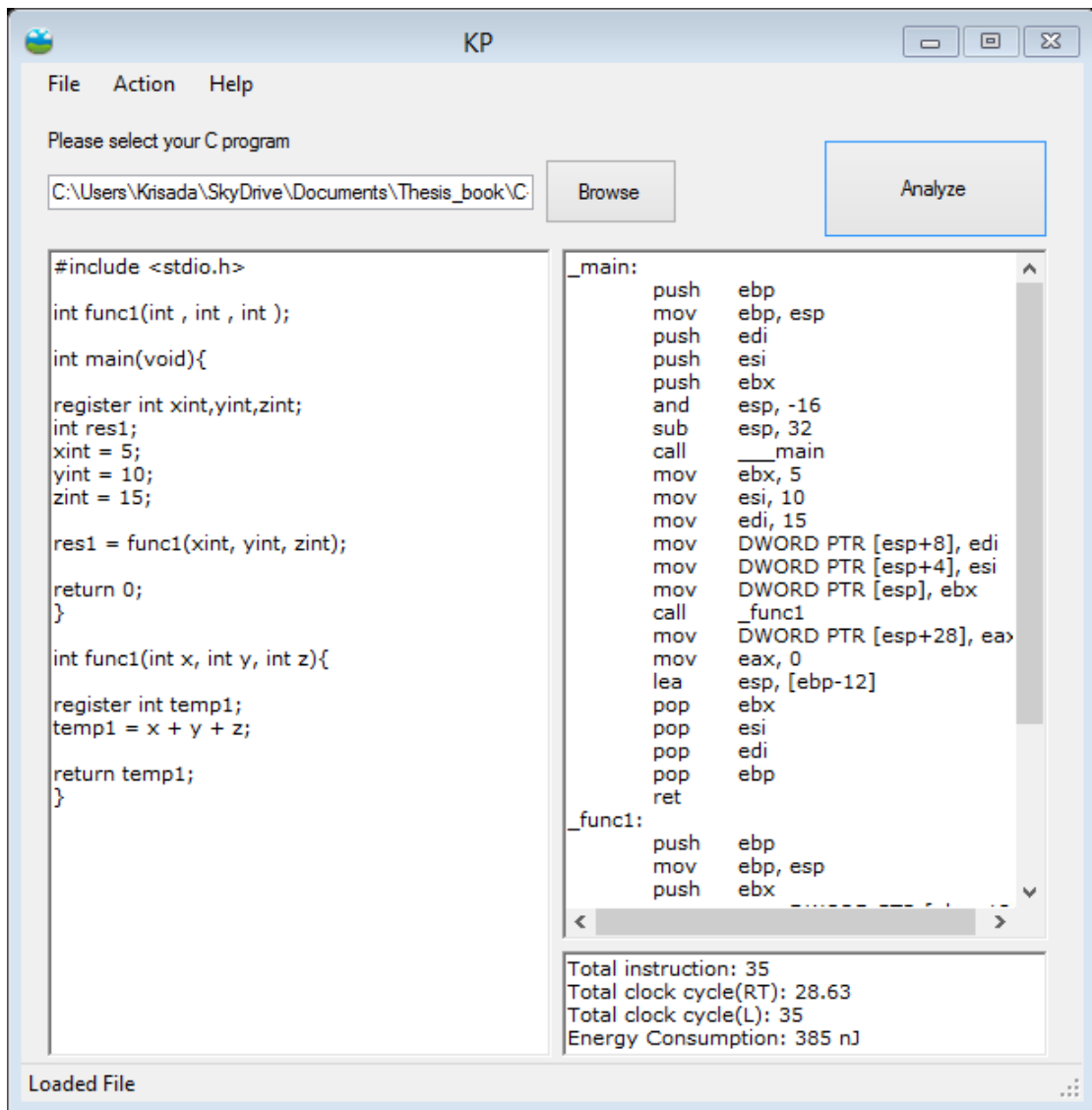


Figure 4.8 KP running case study 1 – register variable

The C program in Figure 4.8, contained 35 instructions that utilized 28.63 clock cycles (RT) and 35 clock cycles (L). Energy consumption was approximately 385 nJ.

4.3 Second case study (2) - Repeated Function calls

The second case, case (2) was conducted in the same manner the previous case. The C program using **local variables** was first tested. Then variables using shared variable was performed, followed by register variables.

The screenshot shows the KP IDE interface. The top menu bar includes 'File', 'Action', and 'Help'. Below the menu is a text box for selecting a C program, with the path 'C:\Users\Krisada\SkyDrive\Documents\Thesis_book\C' and 'Browse' and 'Analyze' buttons. The main window is split into two panes. The left pane contains the C source code:

```
#include <stdio.h>
int func1(int , int , int );
int main(void){
int xint,yint,zint,res1,i;
xint = 5;
yint = 10;
zint = 15;
for(i=0;i<5;i++)
{
res1 = func1(xint, yint, zint);
}
return 0;
}
int func1(int x, int y, int z){
int temp1;
temp1 = x + y + z;
return temp1;
}
```

The right pane shows the assembly code for the program:

```
_main:
push    ebp
mov     ebp, esp
and     esp, -16
sub     esp, 48
call   ___main
mov     DWORD PTR [esp+40], 5
mov     DWORD PTR [esp+36], 10
mov     DWORD PTR [esp+32], 15
mov     DWORD PTR [esp+44], 0
jmp     L2
L3:
mov     eax, DWORD PTR [esp+32]
mov     DWORD PTR [esp+8], eax
mov     eax, DWORD PTR [esp+36]
mov     DWORD PTR [esp+4], eax
mov     eax, DWORD PTR [esp+40]
mov     DWORD PTR [esp], eax
call   _func1
mov     DWORD PTR [esp+28], eax
inc     DWORD PTR [esp+44]
L2:
cmp     DWORD PTR [esp+44], 4
jle    L3
mov     eax, 0
leave
ret
_func1:
<----->
```

At the bottom of the assembly pane, performance statistics are displayed:

```
Total instruction: 124
Total clock cycle(RT): 112.27
Total clock cycle(L): 220
Energy Consumption: 1364 nJ
```

The status bar at the bottom left shows 'Loaded File'.

Figure 4.9 KP running case study 2 – local variable

The C program in Figure 4.9 contained 124 instructions that utilized 112.27 clock cycles (RT) and 220 clock cycles (L). Energy consumption was approximately 1364 nJ. The C code was modified using shared variable as shown in Figure 4.10.

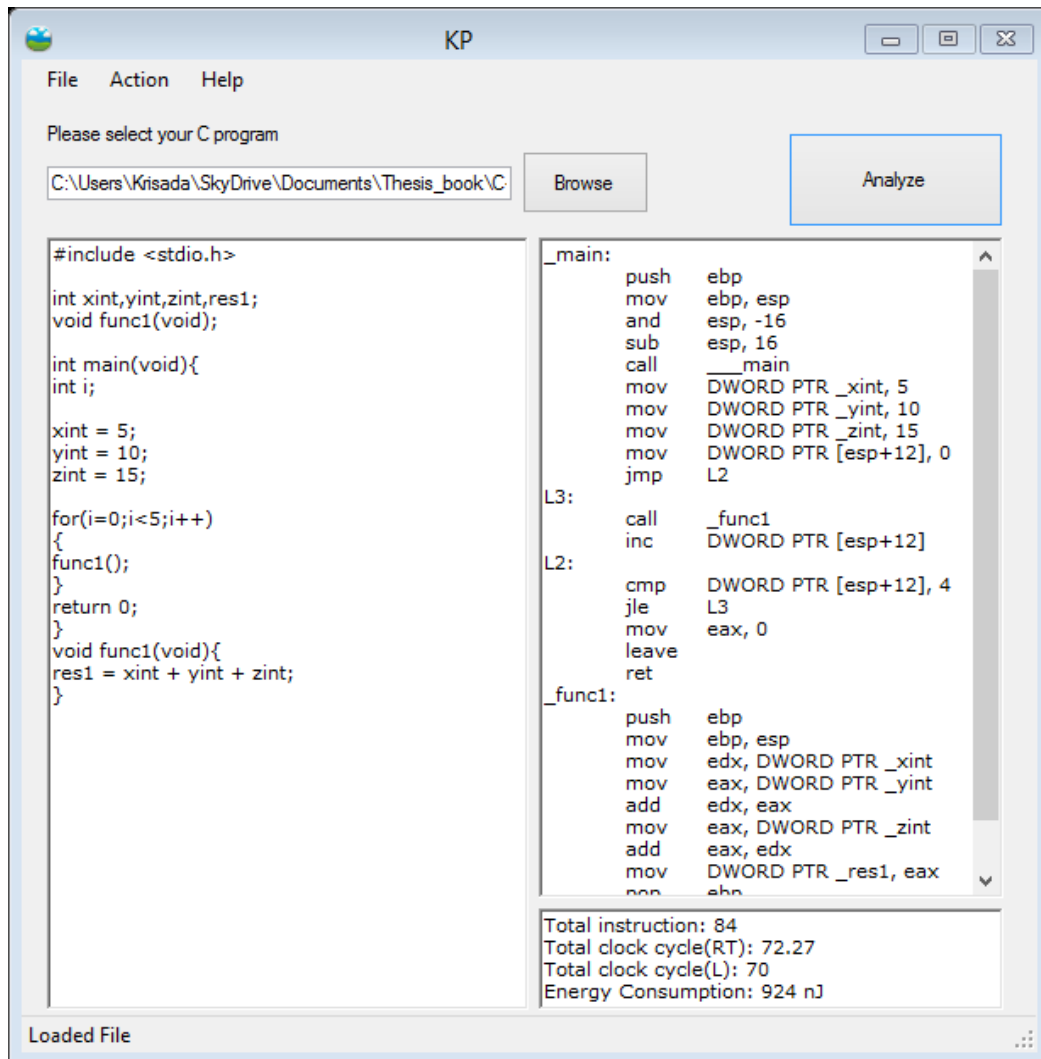


Figure 4.10 KP running case study 2 – shared variable

The C program In Figure 4.10 contained 84 instructions that utilized 72.27 clock cycles (RT) and 70 clock cycles (L). Energy consumption was approximately 924 nJ.

The C code was modified by simply adding the keyword register.

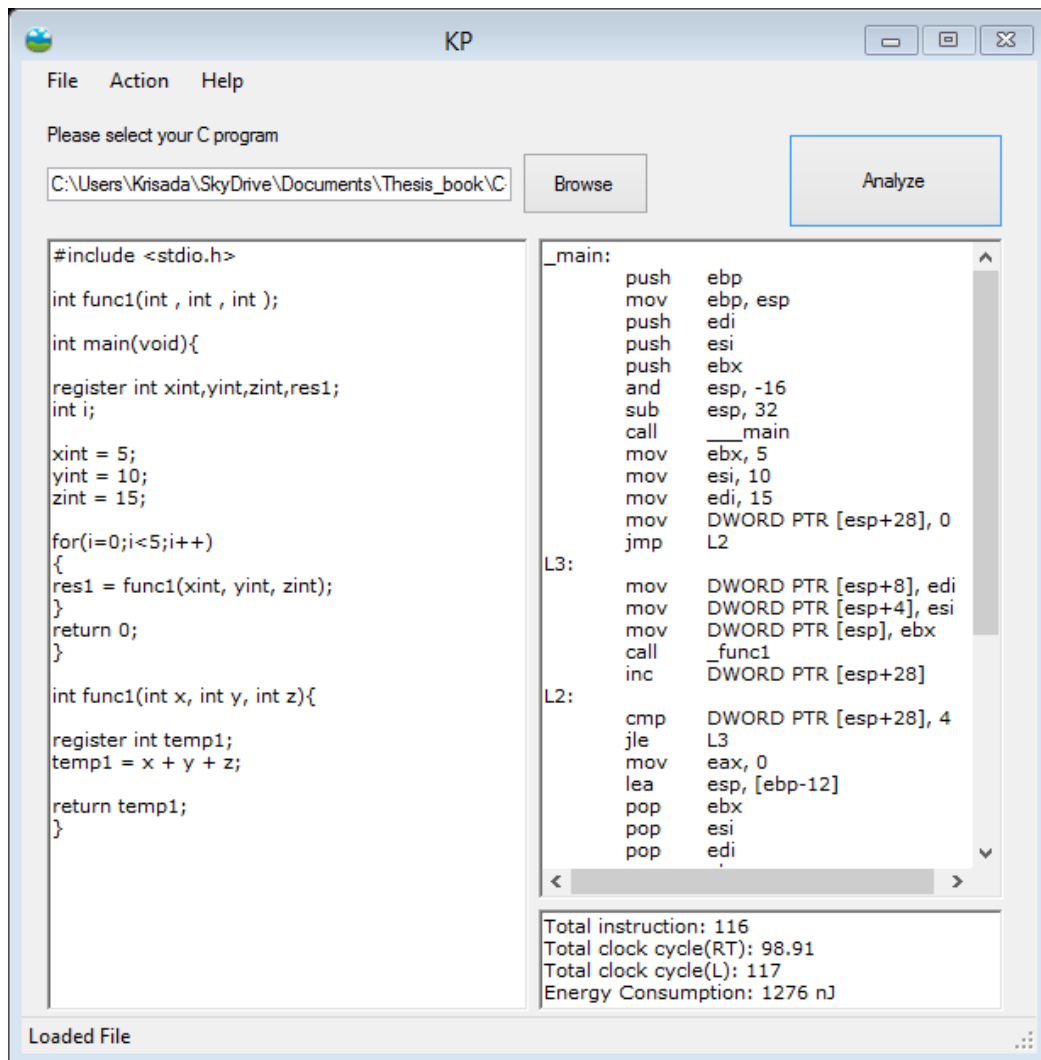


Figure 4.11 KP running case study 2 – register variable

The C program in Figure 4.11 contained 116 instructions that utilized 98.91 clock cycles (RT) and 117 clock cycles (L). Energy consumption was approximately 1276 nJ.

4.4 Third case study (3) - Function calls Function

The third case (3) was carried out in the same manner as previous cases. The C program using **local variables** was first tested. The program was then modified using shared variable and register variables.

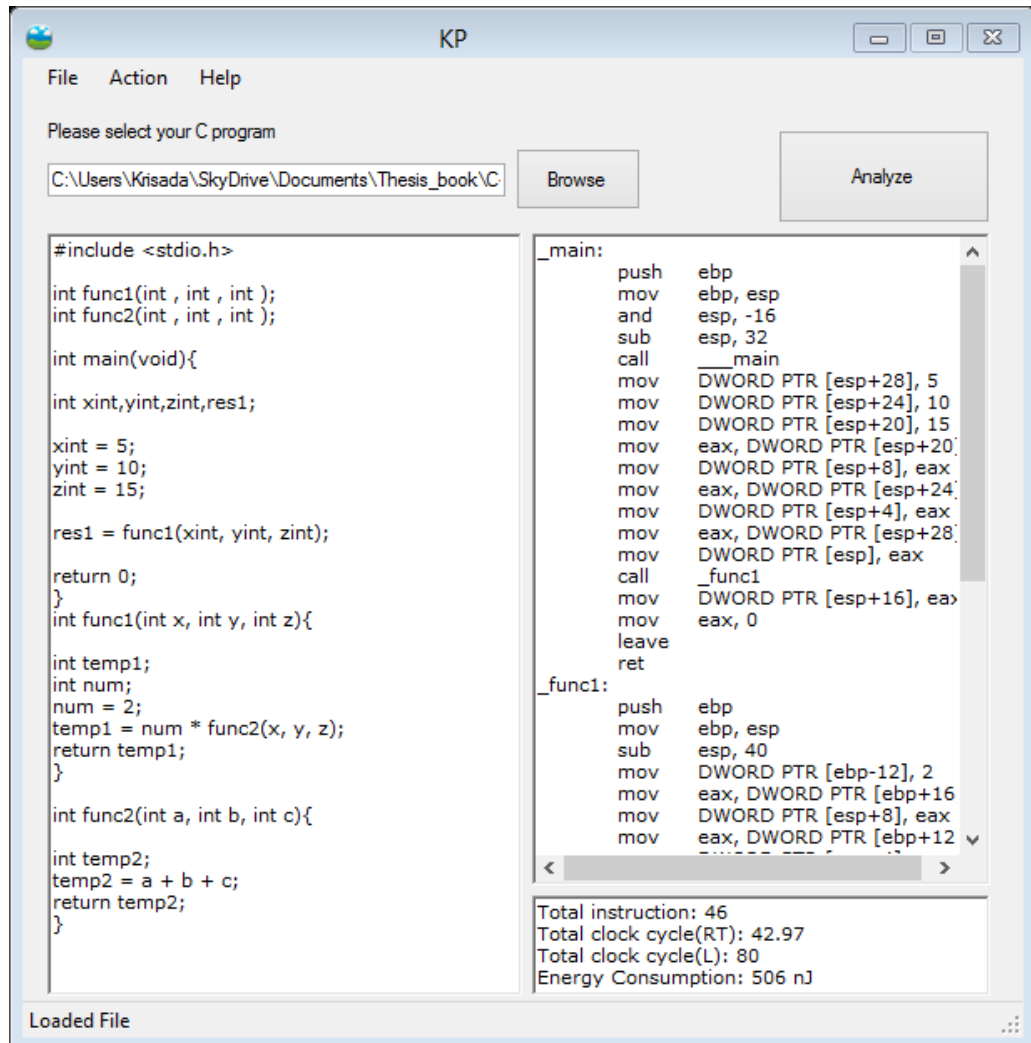


Figure 4.12 KP running case study 3 – local variable

The C program in Figure 4.12 contained 46 instructions that utilized 42.97 clock cycles (RT) and 80 clock cycles (L). Energy consumption was approximately 506 nJ.

The C code was modified using shared variable as shown in Figure 4.13

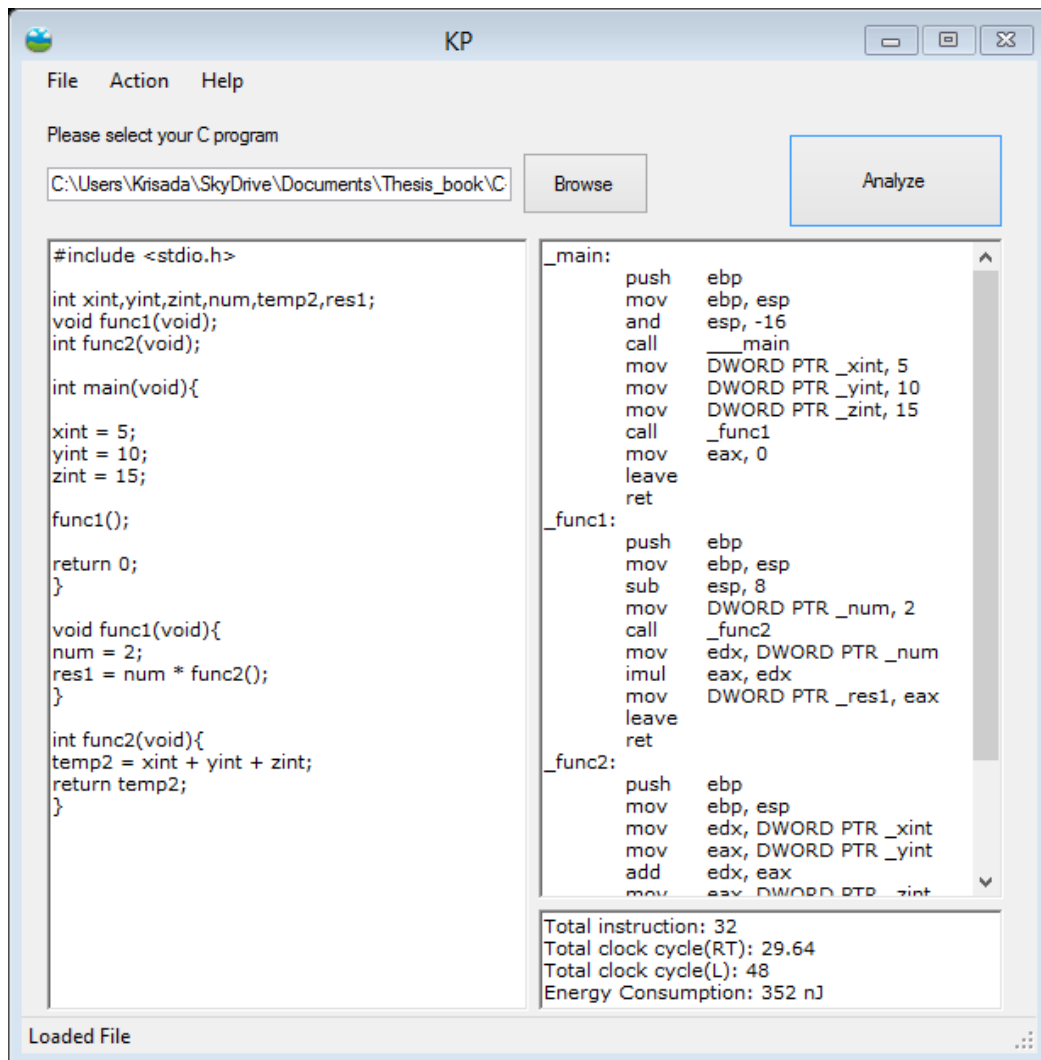


Figure 4.13 KP running case study 3 – shared variable

The C program in Figure 4.13 contained 32 instructions that utilized 29.64 clock cycles (RT) and 48 clock cycles (L). Energy consumption was approximately 352 nJ.

The C code was modified by adding the keyword register in Figure 4.14

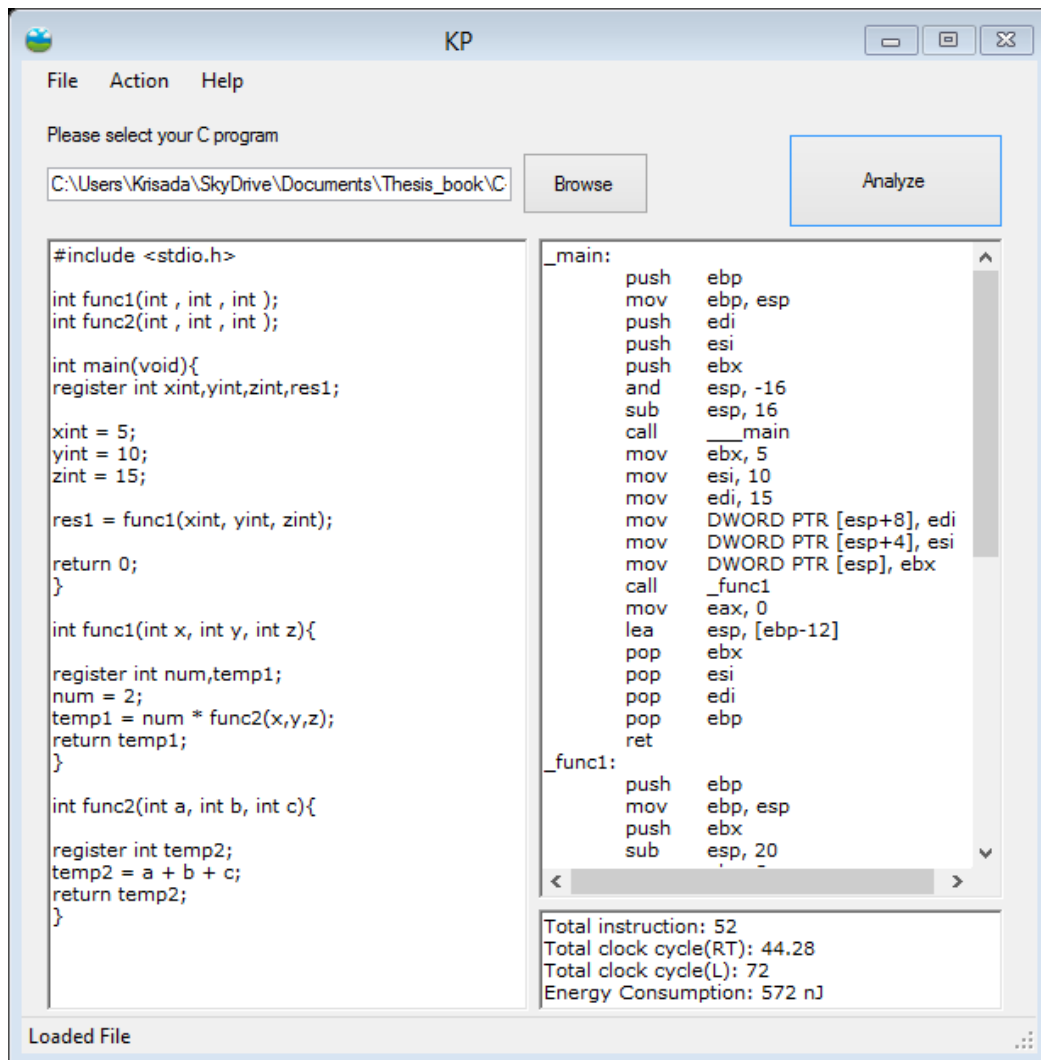


Figure 4.14 KP running case study 3 – register variable

The original C program in the Figure 4.14 contained 52 instructions that utilized 44.28 clock cycles (RT) and 72 clock cycles (L). Energy consumption was approximately 572 nJ.

4.5 Fourth case study (4) – Nested repeated Function calls

The fourth case (4) performed repeated function calls. The C program using **local variables** was first tested and then modified using shared variable and register variables.

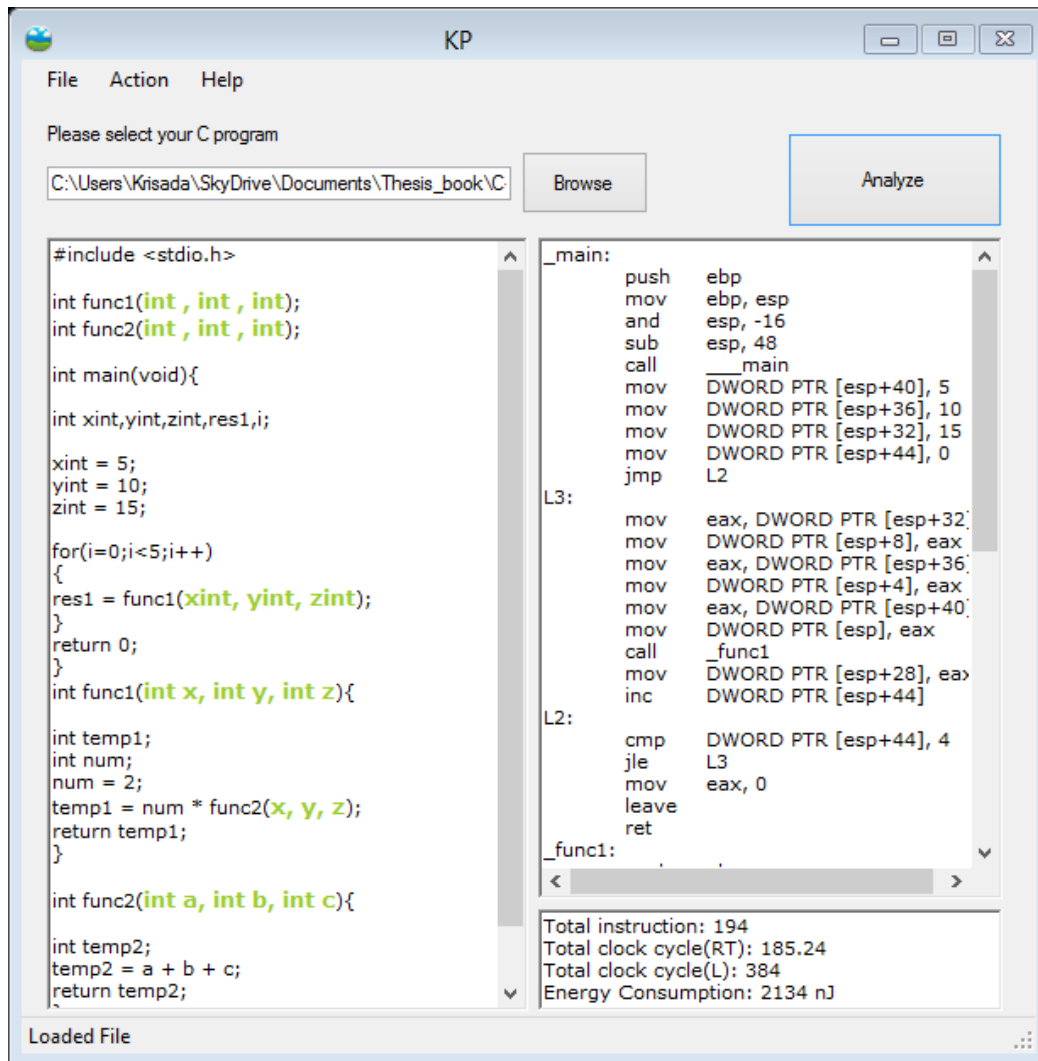


Figure 4.15 KP running case study 4 – local variable

The C program in Figure 4.15 contained 194 instructions that utilized 185.24 clock cycles (RT) and 384 clock cycles (L). Energy consumption was approximately 2134 nJ.

The C code in Figure 4.18 was now modified to use shared variable.

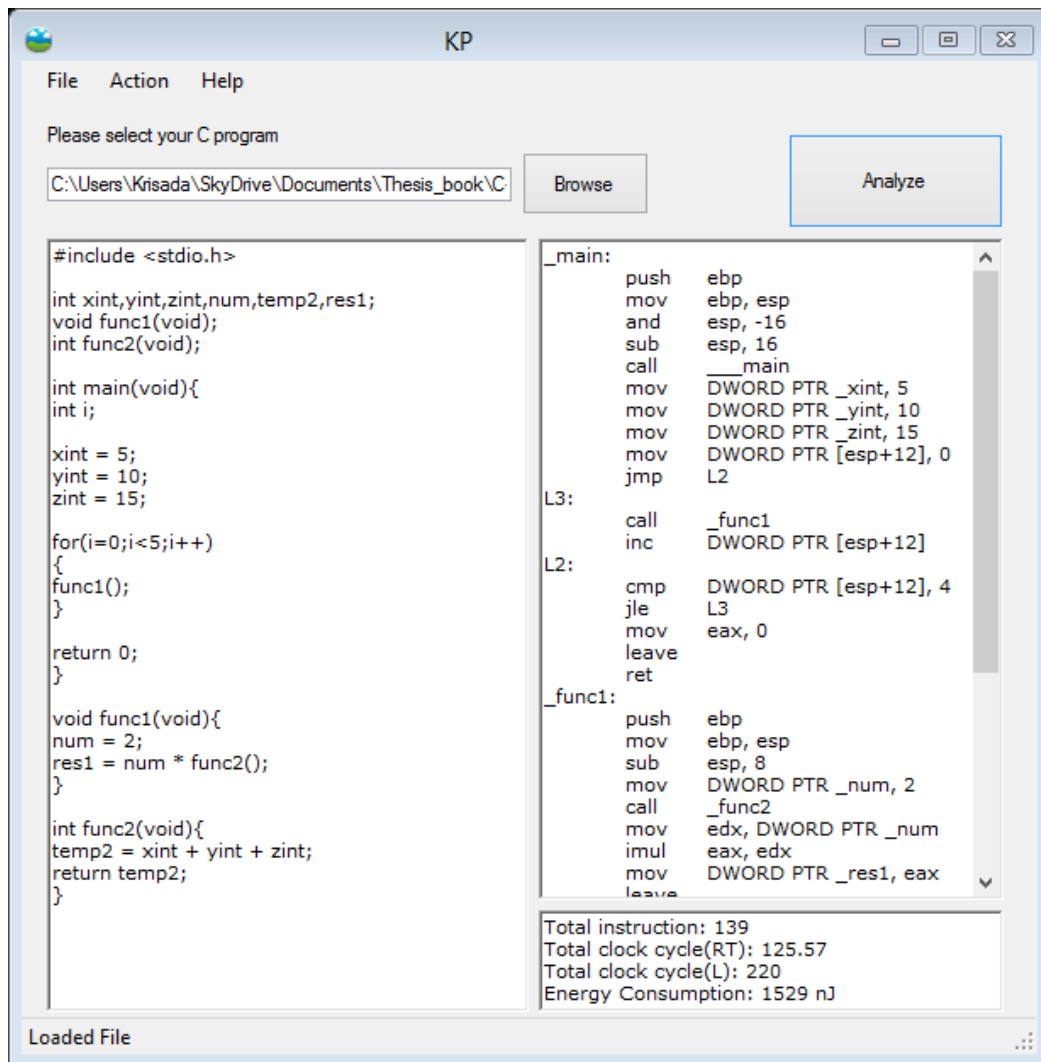


Figure 4.16 KP running case study 4 – shared variable

The C program in Figure 4.16 contained 139 instructions that utilized 125.57 clock cycles (RT) and 220 clock cycles (L). Energy consumption was approximately 1529 nJ.

Finally, the C code was modified using register variables as seen in Figure 4.17.

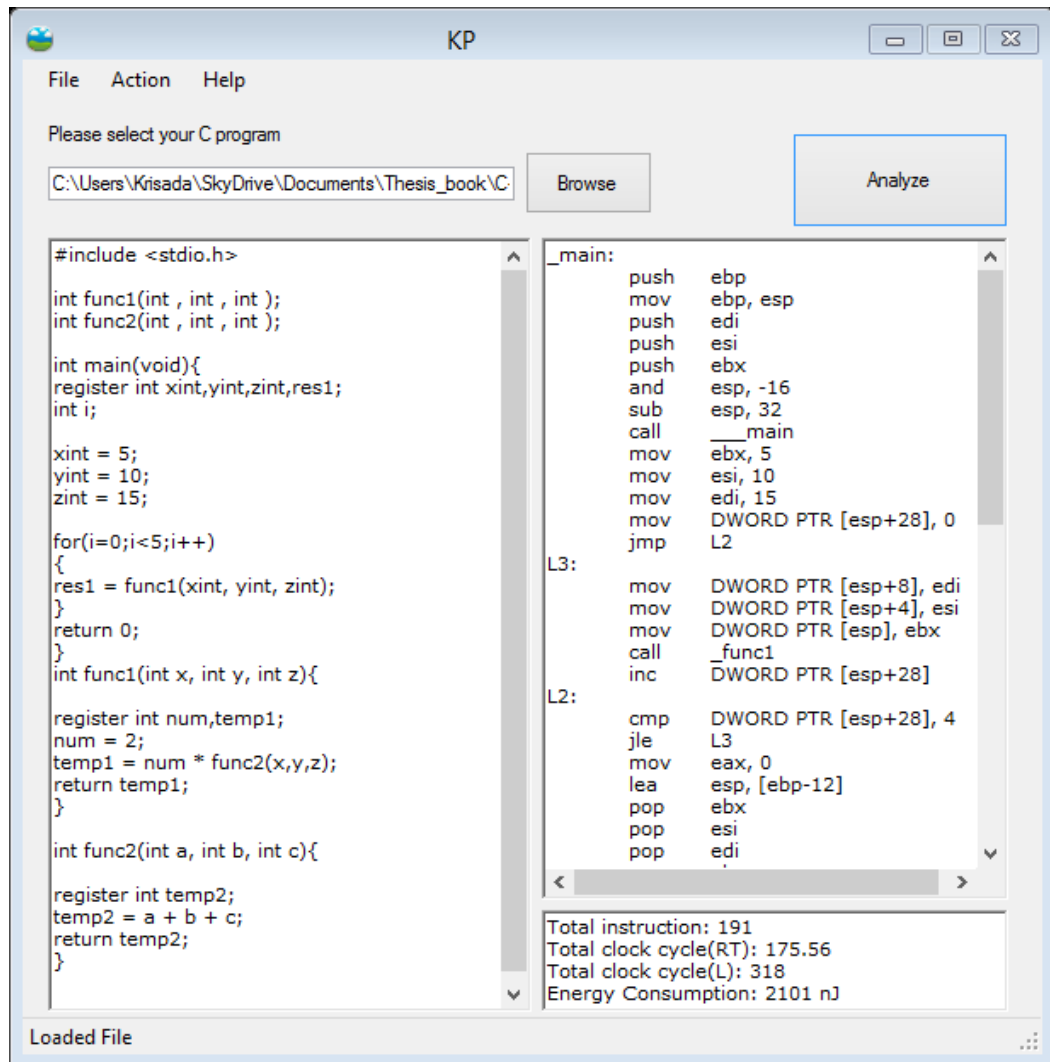


Figure 4.17 KP running case study 4 – register variable

The C program in Figure 4.17 contained 191 instructions that utilized 175.56 clock cycles (RT) and 318 clock cycles (L). Energy consumption was approximately 2101 nJ.

The tables below are the results from the first set experiments, categorized in 5 groups: 1) Number of instructions, 2) Energy consumption by allocation scheme, 3) Number of clock cycle (reciprocal throughput (RT)), 4) Number of clock cycle (latency (L)), and 5) Number of clock cycle (RT + L). The given scenarios are compared in pairwise, namely, Shared variable Vs Local, Register Vs Local, and Shared variable Vs Register).

Table 4.1 Number of instructions

Case	Local	Shared	Register
1	30	21	35
2	124	84	116
3	46	32	52
4	194	139	191

Table 4.2 Energy consumption (nJ) by allocation schemes

Case	Local	Shared	Register	Shared VS Local	Register VS Local	Shared VS Register
1	330	231	385	-30.00 %	+16.67 %	-40.00 %
2	1364	924	1276	-32.26 %	-6.45 %	-27.59 %
3	506	352	572	-30.43 %	+13.04 %	-38.46 %
4	2134	1529	2101	-28.35 %	-1.55 %	-27.23 %

Table 4.3 Number of clock cycle (RT) by allocation schemes

Case	Local	Shared	Register	Shared VS Local	Register VS Local	Shared VS Register
1	27.31	18.98	28.63	-30.50 %	+4.83 %	-33.71 %
2	112.27	72.27	98.91	-35.63 %	-11.90 %	-26.93 %
3	42.97	29.64	43.95	-31.02 %	+2.28 %	-32.56 %
4	185.24	125.57	175.56	-32.21 %	-5.23 %	-28.47 %

Table 4.4 Number of clock cycle (L) by allocation schemes

Case	Local	Shared	Register	Shared VS Local	Register VS Local	Shared VS Register
1	39	25	35	-35.90 %	-10.26 %	-28.57 %
2	220	70	117	-68.18 %	-46.82 %	-40.17 %
3	80	48	70	-40.00 %	-10.00 %	-33.33 %
4	384	220	318	-41.02 %	-14.75 %	-30.82 %

Table 4.5 Number of clock cycle (RT + L) by allocation schemes

Case	Local	Shared	Register	Shared VS Local	Register VS Local	Shared VS Register
1	66.31	43.98	63.63	-33.68 %	-4.04 %	-30.88 %
2	332.27	142.27	215.91	-57.18 %	-35.02 %	-34.11 %
3	122.97	77.64	113.95	-36.86 %	-7.34 %	-31.86 %
4	569.24	345.57	493.56	-39.29 %	-13.29 %	-29.98 %

Results from the Table 4.1 shows that in case (1) - function calls, shared variable utilizes only a few number of instructions (21) while register variable uses a larger number instructions (35). In case (4) – repeated function calls to function, shared variable utilizes fewer instructions than others (139).

In Table 4.2, shared variable of case (1) exhibits a sizable savings (-30%), while register variable shows a slightly higher consumption (+16.67%) than that of the local variable. When comparing shared variable with register variable, it can be seen that shared variable consumes less energy than register variable (-40%).

The results are different for repeated function calls (case 2), where shared variable saves energy consumption (-32.26%), and register variable consumes less energy than the original local variable (-6.45%), and shared variable saves more energy (-27.59%) compared to register variable. As programs become more complicated, savings on energy consumption are even more noticeable. The function calls to other

functions (case 3) exhibits such benefits. Shared variable saves over -30.43% when compared to local variable, while the number on register variable shows slightly higher consumption (+13.04%) than local variable. Shared variable saves -38.46% energy compared to register variable. For repeated function calls to other functions (case 4), the numbers shows that shared variable uses -28.35%, while register variable uses -1.55% and -27.23% when comparing shared variable with register variable.

Table 4.3 summarizes all the statistics taking only clock cycle (RT) factor into account. Similar to previous tables, all variables are compared. In case (1), shared variable is faster than local variable by -30.50%, register is slightly slower than local variable by about +4.83%, while shared variable is faster than register variable by -33.71%. When looking at case 4 which contains repetitions, the shared variable is still faster than local variable (-32.21%), while register variable is faster than local variable (-5.23%)

Table 4.4 shows clock cycle latency (L). In case (1), local variable has the highest latency at about 39 cycles. Register is next at 35 cycles, and shared variable is last at 25 cycles. Shared variable have fewer latency clock cycles (-35.90%) than local variable, and register variable is even less at 10.26%. The results show that shared variable is -28.57% when compared with register.

Table 4.5 shows the total clock cycle (L + RT). In case (1), Shared variable uses -33.68% less than local variable and less than register -30.88%. Register variable also expends less clock cycles (-4.04%) than local variable.

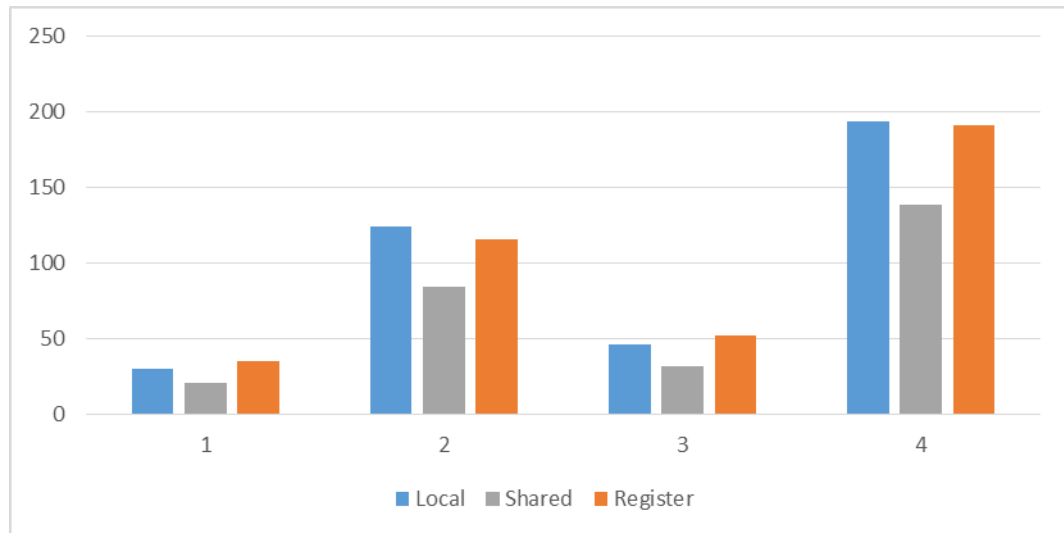


Figure 4.18 Comparison of number of instruction (Y) for all cases (X)

Figure 4.18 plots the results of Table 4.1 that shared variable surpasses other schemes in all cases.

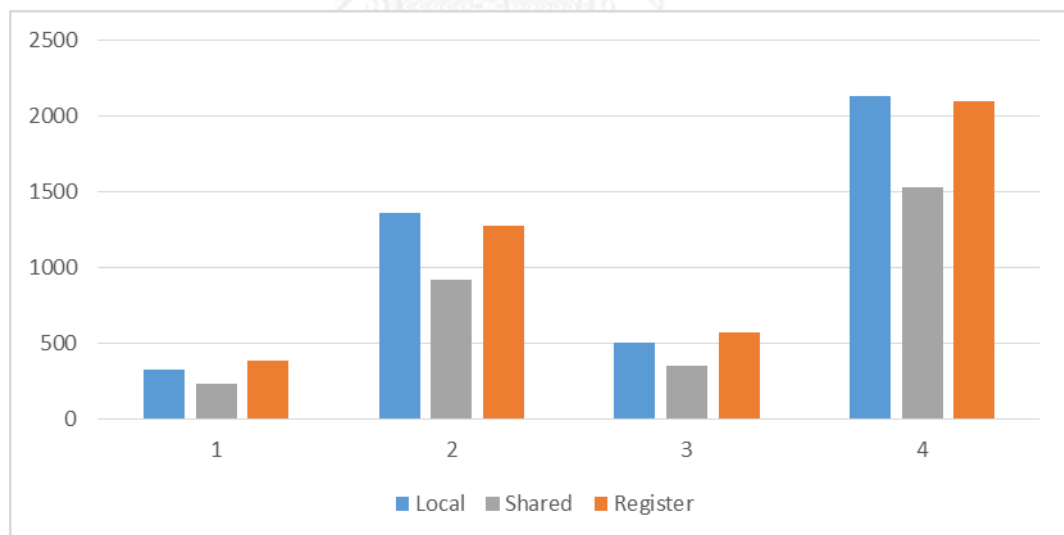


Figure 4.19 Comparison of energy consumption (Y) for all cases (X)

Figure 4.19 plots the results of Table 4.2 that shared variable consumes the lowest all cases, while the original local variable takes the highest in case (1) and (3).



Figure 4.20 Comparison of clock cycle (RT) (Y) for all cases (X)

Figure 4.20 corresponds to Table 4.3. Note that local variable has the highest clock values in case (1) and (3), while shared variable has the lowest values in every case.



Figure 4.21 Comparison of clock cycle (L) (Y) for all cases (X)

Figure 4.21 corresponds to Table 4.4. Note that local variable has the highest clock latency values in all cases.

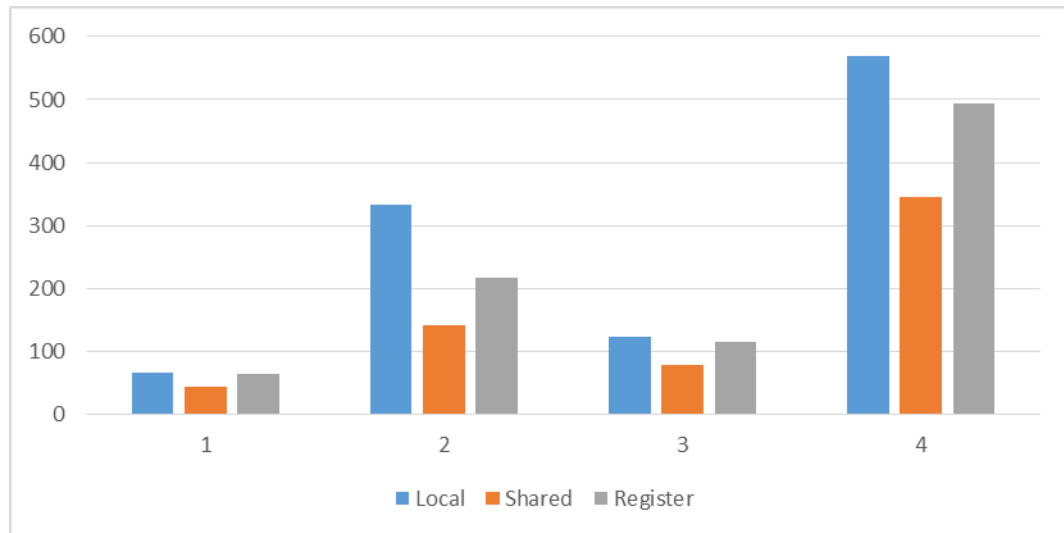


Figure 4.22 Comparison of clock (RT + L) (Y) for all cases (X)

Figure 4.22 corresponds to Table 4.5. Note that local variable has the highest RT+L values in all cases.

Table 4.6 Showing all results from 4 cases

	1			2			3			4		
	local	shared	register	local	shared	register	local	shared	register	local	shared	register
Total Instruction	30	21	35	124	84	116	46	32	52	194	139	191
Total Energy Consumption	330	231	385	1364	924	1276	506	352	572	2134	1529	2101
Total Clock (RT)	27.31	18.98	28.63	112.27	72.27	98.91	42.97	29.64	43.95	185.24	125.57	175.56
Total Clock (L)	39	25	35	220	70	117	80	48	70	384	220	318
Total Clock (RT + L)	66.31	43.98	63.63	332.27	142.27	215.91	122.97	77.64	113.95	569.24	345.57	493.56

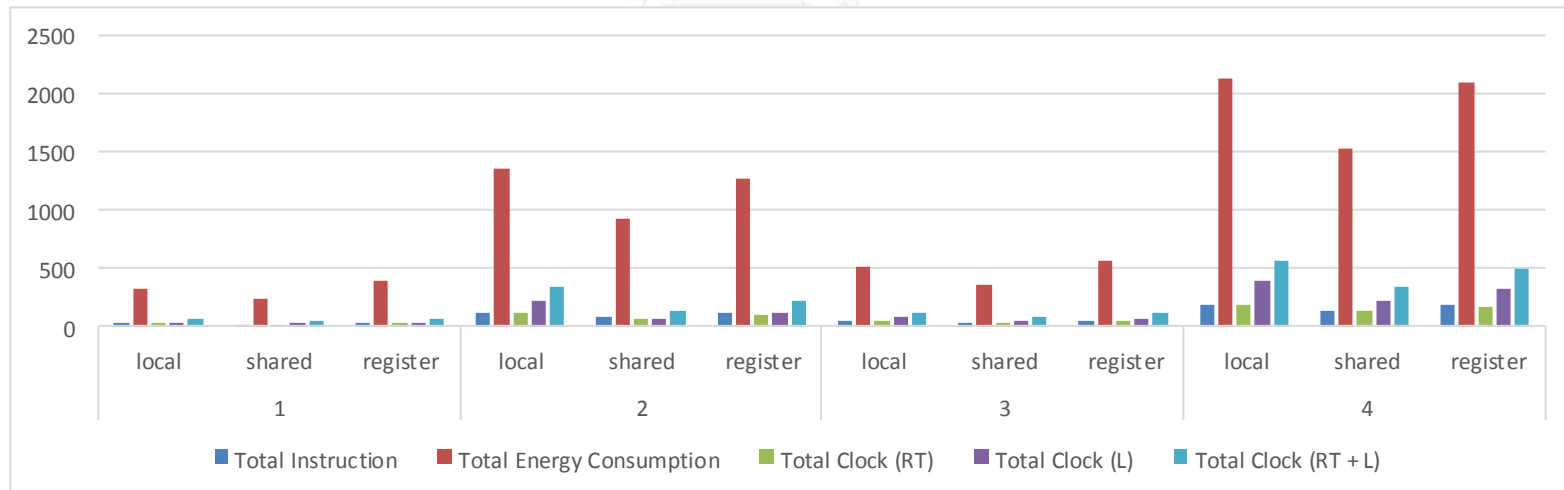


Figure 4.23 corresponds to Table 4.6

4.6 Discussion

The results obtained from the KP tool did not come as a surprise, as they were well-established programming facts. The findings merely reinstated local, register, and shared variable scoping principles. From the findings, shared variable reduced energy consumption by approximately 30% in simple function calls, and around 30% in nested function calls. Register variables, on the other hand, were effective only when repetitive accesses were required. However, it was not as efficient as shared variable. Energy consumption comparison between shared variable and register variables reveals that the former saved approximately 40% and 27% in simple function calls and nested function calls, respectively.

Table 4.6 and Figure 4.23 show an interesting finding that is rarely stated in the literature regarding total energy consumption of different methods of variable scoping. This experiment found that local variable consumed the most energy. From the instruction level, it was apparent that stack process took considerable clock cycles which resulted in high latency and energy consumption.

The shortfall of shared variable is due to memory access protection as it violates information hiding principles in Software Engineering. The side effect of shared access is what programmers should heed and practice with care. This is a tradeoff of using shared variable over local variable during software design. Programmers have to choose which direction will go, either saving energy consumption or preserve software engineering principle. As such, they can choose proper variable type to access memory. An straightforward example is the use of global static variable to make a variable visible, yet prevents accidental access by code due to naming conflicts.

Another example is register variable that is often applied to repetitive accesses of memory to reduce the energy consumption on stack pushes and pops in typical parameter passing mechanisms.

To further reatfirm the above finding, the second set of programs were experimented. They were more complicated, more parameter and more loops. The tables below show the results from the second set of experiments. All sample C code can be found in the Appendix A.

Table 4.7 Number of instructions (set 2)

Case	Local	Shared	Register
1	37	29	41
2	313	243	290
3	56	29	61
4	495	374	481

Table 4.8 Energy consumption by allocation scheme (set 2)

Case	Local	Shared	Register	Shared VS Local	Register VS Local	Shared VS Register
1	407	319	451	-21.62 %	10.81 %	-29.27 %
2	3443	2673	3190	-22.36 %	-7.35 %	-16.21 %
3	616	319	671	-48.21 %	8.93 %	-52.46 %
4	5445	4114	5291	-24.44 %	-2.83 %	-22.25 %

Table 4.9 (RT) Number of clock cycle by allocation scheme (set 2)

Case	Local	Shared	Register	Shared VS Local	Register VS Local	Shared VS Register
1	30.13	23.47	31.12	-22.10 %	3.29 %	-24.58 %
2	243.39	180.09	215.08	-26.01 %	-11.63 %	-16.27 %
3	48.12	33.46	49.77	-30.47 %	3.43 %	-32.77 %
4	414.29	287.59	385.88	-30.58 %	-6.86 %	-25.47 %

Table 4.10 (L) Number of Clock cycle by allocation scheme (set 2)

Case	Local	Shared	Register	Shared VS Local	Register VS Local	Shared VS Register
1	62	44	54	-29.03 %	-12.90 %	-18.52 %
2	559	389	472	-30.41 %	-15.56 %	-17.58 %
3	98	60	88	-38.78 %	-10.20 %	-31.82 %
4	892	603	765	-32.40 %	-14.24 %	-21.18 %

Table 4.11 (RT + L) Number of clock cycle (set 2)

Case	Local	Shared	Register	Shared VS Local	Register VS Local	Shared VS Register
1	92.13	67.47	85.12	-26.77 %	-7.61 %	-20.74 %
2	802.39	569.09	687.08	-29.08 %	-14.37 %	-17.17 %
3	146.12	93.46	137.77	-36.04 %	-5.71 %	-32.16 %
4	1306.29	890.59	1150.88	-31.82 %	-11.90 %	-22.62 %

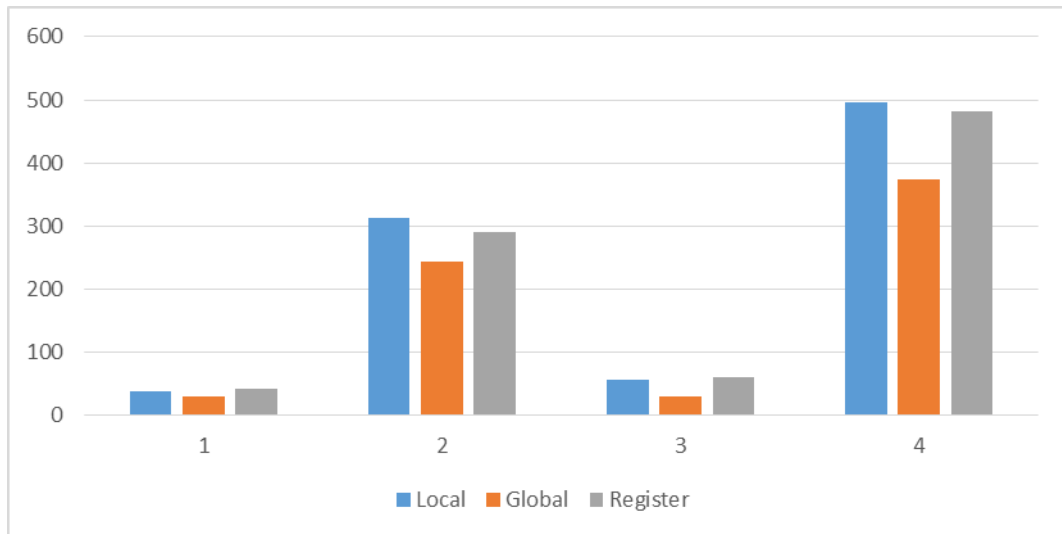


Figure 4.24 Comparison of number of instruction (Y) for all cases (X)

Figure 4.23 corresponds to Table 4.6. Note that shared variable has the lowest values in every case.

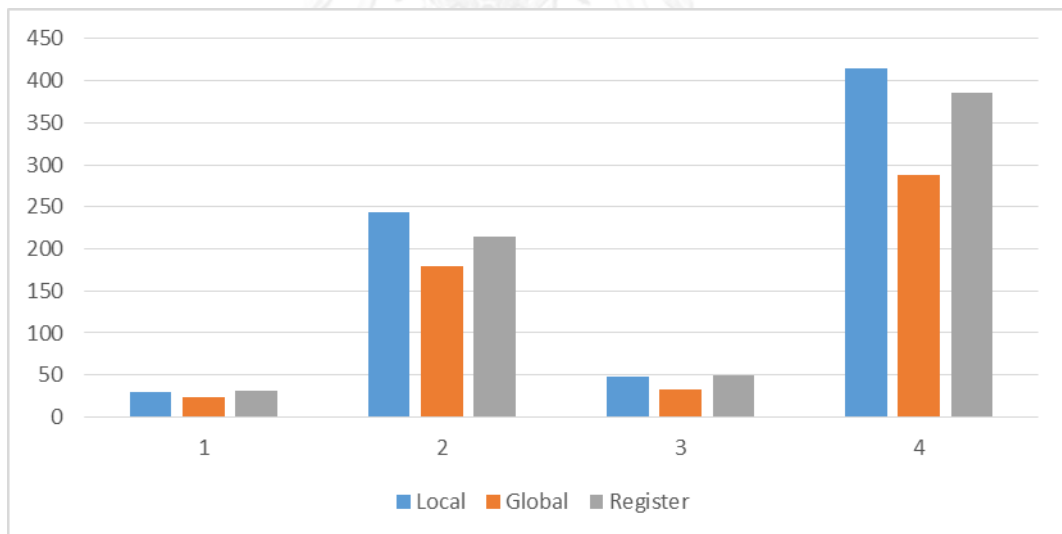


Figure 4.25 Comparison of energy consumption (Y) for all cases (X)

Figure 4.23 corresponds to Table 4.7. Note that shared variable has the lowest values in every case.

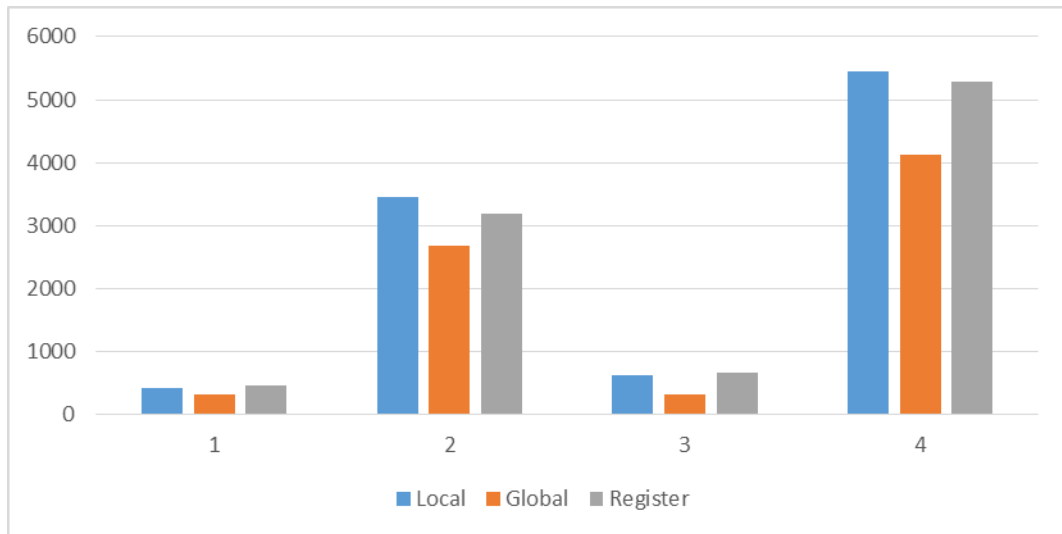


Figure 4.26 Comparison of clock cycle (RT) for all cases (X)

Figure 4.23 corresponds to Table 4.8. Note that register variable has lower values than local variable for cases (2) and (4).



Figure 4.27 Comparison of clock cycle (L) (Y) for all cases (X)

Figure 4.23 corresponds to Table 4.9. Note that local variable has the highest values in every case.

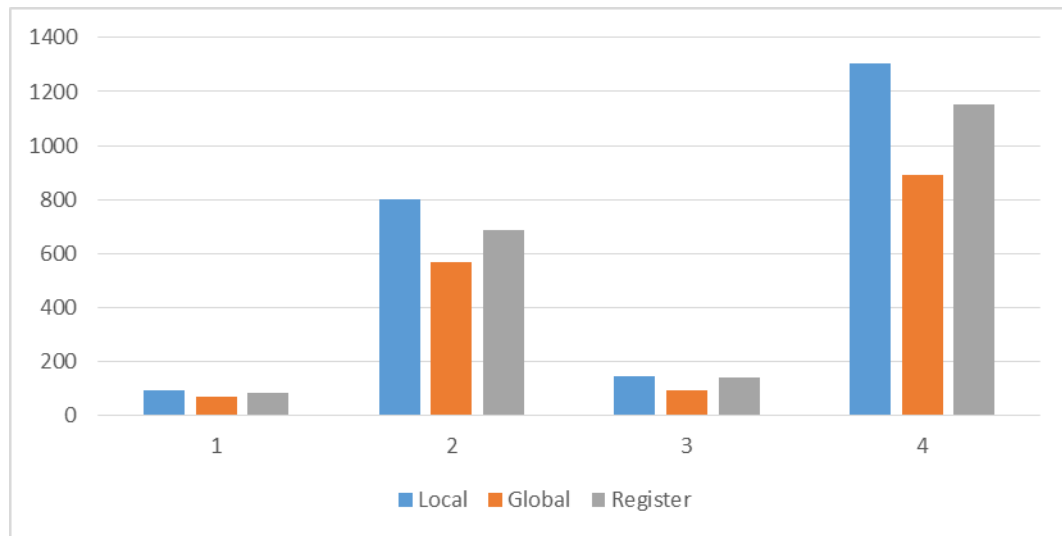


Figure 4.28 Comparison of clock cycle (L+RT) (Y) for all cases (X)

Figure 4.23 correspond to Table 4.10. Note that local variable has the highest values in every case. All the results from set 2 are similar to the previous discussion.

Chapter 5

Conclusion and future work

5.1 Conclusion and future work

This thesis investigates energy consumption and clock cycle used by local variable, register variable, and shared variable in C programs. Experiments were conducted on 24 C sample programs in 2 scenarios and 4 case studies by means of the KP tool. The results shows that shared variable significantly consumes the least energy over local and register variables for all 4 cases. Nevertheless, the gains from shared variable could possibly be offset by violation penalties of “good” software engineering practices, e.g., side effects, information hiding, portability, etc. The issue at hand is whether software engineering or energy consumption is crucial to producing theoretically sound or environmentally conscious products.

The KP tool greatly helps identify code fragment to reduce the energy consumption from the programming point of view. However, it is currently limited to basic analysis. Future study will improve the tool to be able to analyze complicated C programs in wider research contexts.

REFERENCES

1. [cited 2014 April]; Available from: http://en.wikipedia.org/wiki/World_energy_consumption.
2. ; Available from: <http://www.migsmobile.net/2010/01/12/evolution-of-mobile-device-uses-and-battery-life>.
3. *Mobile Future of Connected Devices*. 2013.
4. Lee, S., et al., *An Accurate Instruction-Level Energy Consumption Model for Embedded RISC Processors*, in *Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems*. 2001, ACM: Snow Bird, Utah, USA. p. 1-10.
5. Nouredine, A., A. Bourdon, and L. Seinturier, *A Preliminary Study of the Impact of Software Engineering on GreenIT*.
6. Flinn, J. and M. Satyanarayanan. *PowerScope: a tool for profiling the energy usage of mobile applications*. in *Mobile Computing Systems and Applications, 1999. Proceedings. WMCSA '99. Second IEEE Workshop on*. 1999.
7. Amsel, N. and B. Tomlinson, *Green Tracker: A Tool for Estimating the Energy Consumption of Software*. 2010.
8. Do, T., S. Rawshdeh, and W. Shi, *pTop: A Process-level Power Profiling Tool*.
9. Shih, E., P. Bahl, and M.J. Sinclair, *Wake on wireless: an event driven energy saving strategy for battery operated devices*, in *Proceedings of the 8th annual international conference on Mobile computing and networking*. 2002, ACM: Atlanta, Georgia, USA. p. 160-171.
10. Mike Tien-Chien, L. and V. Tiwari. *A memory allocation technique for low-energy embedded DSP software*. in *Low Power Electronics, 1995., IEEE Symposium on*. 1995.
11. Binkley, D., *Source Code Analysis: A Road Map*, in *2007 Future of Software Engineering*. 2007, IEEE Computer Society. p. 104-119.
12. Sutherland, B., *Beginning Android C++ Game development*. 2013.

13. Tsai, Y.-T., et al., *Mobile visual computing in C++ on Android*, in *ACM SIGGRAPH 2013 Mobile*. 2013, ACM: Anaheim, California. p. 1-1.
14. Wagner, T.A., et al., *Accurate static estimators for program optimization*, in *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*. 1994, ACM: Orlando, Florida, USA. p. 85-96.
15. Skaller, J.M., *A proposal for NESTED FUNCTIONS*. 1993.
16. Yanbing, L. and J. Henkel. *A framework for estimating and minimizing energy dissipation of embedded HW/SW systems*. in *Design Automation Conference, 1998. Proceedings*. 1998.
17. Tiwari, V., S. Malik, and A. Wolfe, *Power analysis of embedded software: a first step towards software power minimization*. *Very Large Scale Integration (VLSI) Systems*, IEEE Transactions on, 1994. 2(4): p. 437-445.
18. Linn, C., et al., *Stack analysis of x86 executables*.
19. Reps, T. and G. Balakrishnan, *Improved memory-access analysis for x86 executables*, in *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th international conference on Compiler construction*. 2008, Springer-Verlag: Budapest, Hungary. p. 16-35.
20. Sestoft, P., *Replacing function parameters by global variables*, in *Proceedings of the fourth international conference on Functional programming languages and computer architecture*. 1989, ACM: Imperial College, London, United Kingdom. p. 39-53.
21. Davidson, J.W. and D.B. Whalley, *Methods for saving and restoring register values across function calls*. *Softw. Pract. Exper.*, 1991. 21(2): p. 149-165.
22. Grochowski, E. and M. Annavaram, *Energy per Instruction Trends in Intel® Microprocessors 2006*.
23. Fog, A., *Instruction Tables Lists of Instruction Latencies, Throughputs and Micro Operation Breakdowns for Intel, AMD and VIA CPUs*, C.U.C.o. Engineering, Editor.
24. *MinGW Minimalist GNU for Windows*. 2013; Available from: <http://www.mingw.org/>.



APPENDIX

จุฬาลงกรณ์มหาวิทยาลัย
CHULALONGKORN UNIVERSITY

Appendix A

Set 2 – Sample programs

Program 1

```
#include<stdio.h>
int findInterest(int,int,int);
int main()
{
int amount,rate,time,res;
amount = 500;
rate = 5;
time = 2;
res = findInterest(amount,rate,time);
}
int findInterest(int a, int r, int t){
int si;
si = (a * r * t)/100;
return si;
}
```

Program 2

```
#include<stdio.h>
void findInterest(void);
int amount,rate,time,si;
int main()
{
int res;
amount = 500;
rate = 5;
time = 2;
findInterest();
}
void findInterest(){
si = (amount * rate * time)/100;
}
```

Program 3

```
#include<stdio.h>
int findInterest(int,int,int);
int main()
{
register int amount,rate,time;
int res;
amount = 500;
rate = 5;
time = 2;
res = findInterest(amount,rate,time);
}
int findInterest(int a, int r, int t){
int si;
si = (a * r * t)/100;
return si;
}
```

Program 4

```
#include<stdio.h>
int findInterest(int,int,int);
int main()
{
int amount,rate,time,res,i;
amount = 500;
rate = 5;
time = 2;
for(i=0;i<10;i++){
    res = findInterest(amount,rate,time);
}
}
int findInterest(int a, int r, int t){
int si;
si = (a * r * t)/100;
return si;
}
```


Program 5

```
#include<stdio.h>
void findInterest(void);
int amount,rate,time,si;
int main()
{
int i;
amount = 500;
rate = 5;
time = 2;
for(i=0;i<10;i++){
    findInterest();
}
}
void findInterest(){
si = (amount * rate * time)/100;
}
```

Program 6

```
#include<stdio.h>
int findInterest(int,int,int);
int main()
{
register int amount,rate,time;
int res,i;
amount = 500;
rate = 5;
time = 2;
for(i=0;i<10;i++){
    res = findInterest(amount,rate,time);
}
}
int findInterest(int a, int r, int t){
register int si;
si = (a * r * t)/100;
return si;
}
```

Program 7

```

#include<stdio.h>
int findInterest(int,int,int,int);
int amount(int,int);
int main()
{
int rate,time,val1,val2,res;
rate = 5;
time = 2;
val1 = 300;
val2 = 200;
res = findInterest(rate,time,val1,val2);
}
int findInterest( int r, int t,int v1,int v2){
int si, sum;
sum = amount(v1,v2);
si = (sum * r * t)/100;
return si;
}
int amount (int v1,int v2)
{
int total;
total = v1 + v2 ;
return total;
}

```

Program 8

```

#include<stdio.h>
void findInterest(void);
void amount(void);
int rate,time,val1,val2,total,si;
int main()
{
int res;
rate = 5;
time = 2;
val1 = 300;

```

```

val2 = 200;
findInterest();
}
void findInterest( ){
amount();
si = (total * rate * time)/100;
}
void amount ()
{
total = val1 + val2 ;
}

```

Program 9

```

#include<stdio.h>
int findInterest(int,int,int,int);
int amount(int,int);
int main()
{
register int rate,time,val1,val2;
int res;
rate = 5;
time = 2;
val1 = 300;
val2 = 200;
res = findInterest(rate,time,val1,val2);
}
int findInterest( int r, int t,int v1,int v2){
int si, sum;
sum = amount(v1,v2);
si = (sum * r * t)/100;
return si;
}
int amount (int v1,int v2)
{
int total;
total = v1 + v2 ;
return total;
}

```

Program 10

```

#include<stdio.h>
int findInterest(int,int,int,int);
int amount(int,int);
int main()
{
int rate,time,val1,val2,res,i;
rate = 5;
time = 2;
val1 = 300;
val2 = 200;
for(i=0;i<10;i++){
    res = findInterest(rate,time,val1,val2);
}
}
int findInterest( int r, int t,int v1,int v2){
int si, sum;
sum = amount(v1,v2);
si = (sum * r * t)/100;
return si;
}
int amount (int v1,int v2)
{
int total;
total = v1 + v2 ;
return total;
}

```

CHULALONGKORN UNIVERSITY

Program 11

```

#include<stdio.h>
void findInterest(void);
void amount(void);
int rate,time,val1,val2,total,si,i;
int main()
{
int res;
rate = 5;

```

```

time = 2;
val1 = 300;
val2 = 200;
for(i=0;i<10;i++){
    findInterest();
}
}
void findInterest( ){
amount();
si = (total * rate * time)/100;
}
void amount ( )
{
total = val1 + val2 ;
}

```

Program 12

```

#include<stdio.h>
int findInterest(int,int,int,int);
int amount(int,int);
int main()
{
register int rate,time,val1,val2;
int res,i;
rate = 5;
time = 2;
val1 = 300;
val2 = 200;
for(i=0;i<10;i++){
    res = findInterest(rate,time,val1,val2);
}
}
int findInterest( int r, int t,int v1,int v2){
register int si, sum;
sum = amount(v1,v2);
si = (sum * r * t)/100;
return si;
}

```

```
int amount (int v1,int v2)
{
register int total;
total = v1 + v2 ;
return total;
}
```



จุฬาลงกรณ์มหาวิทยาลัย
CHULALONGKORN UNIVERSITY

Appendix B

Conference Paper - 2014 International Conference on Computer, Network Security and Communication Engineering (CNSCE 2014) ISBN: 978-1-60595-167-6 p.712-716

2014 International Conference on Computer, Network Security and Communication Engineering (CNSCE 2014)
ISBN: 978-1-60595-167-6

Reducing Energy Consumption in C Programs by Variable Reallocation

Krisada Samrittianusorn and Peraphon Sophatsathit

Advanced Virtual and Intelligent Computing (AVIC) Center

Department of Mathematics and Computer Science, Faculty of Science

Chulalongkorn University, Bangkok, 10330, Thailand

krisada.sa@student.chula.ac.th

Keywords: Energy consumption, Variable reallocation, Instruction level, Memory access protection.

Abstract. Energy consumption around the world increases exponentially. One of the causes to blame is electronic devices such as personal computers, embedded devices, and smartphones. To reckon with reducing energy consumption involves efficient hardware and software. This research focuses on the software part, in particular, how to write a program that is energy efficient. The proposed technique is based primarily on local variable reallocation in C programs to exploit the advantages of shared memory and register variable. We analyze the amount of energy consumed at instruction level. Our findings reveal that shared memory is the best choice at the price of memory access protection. The benefits are fewer redundant allocations and memory accesses, thereby less energy will be consumed.

Introduction

Nowadays, energy consumption is one the most concerned issue world-wide. Various electronic devices such as personal computers, smartphones, and embedded devices are the major culprits of high energy sources. Computer programs will play a central role in all relating applications. What follows is the enormity of energy consumed by these devices. There are many ways to reduce the energy consumption on computers attributed by hardware and software. Studies have shown that software is a principal factor on energy consumption in computer systems [1]. Unfortunately, some programmers may only concern about running time or resource utilization of the program and ignore about energy perspective.

A typical computer program in execution stores, retrieves, and processes variables such as local variables, shared memories, and register variables. Heavy use of these variables wastes considerable energy. One remedy is reorganization of the original code to properly allocate variables and parameters, thereby balancing the distribution of energy consumption. This research will address such a compelling issue to demonstrate how the problem can be alleviated.

The organization of this paper is as follows. Section 2 discusses related researches on saving energy consumption of computers. Section 3 describes the proposed approach to appropriate suitable variables for energy saving. In Section 4, a small research tool was built for the experimental purpose to identify the energy consumption at instruction level. Section 5 discusses comparative results obtained from the experiment. Some final thoughts on the trade-off and future work are also given.

Related Work

Saving energy on electronic devices has been the focus of today's green technology. Many research endeavors have been carried out which can be classified into 2 types, namely, hardware and software. Various techniques have been attempted to cope with such problems. A simple, effective, and popular technique is turning off power in wireless card when it is not used [4]. From software standpoint, properly managed of memory allocation and access will help reduce the amount of energy consumption [7]. At a finer grained level, Grochowki and Annavaram [3] analyzed energy per

instruction (EPI) based on Intel processor. Tools for instruction power analysis [5] or energy aware that help developers determine the energy consumed by their programs under development [6] are also available. All these techniques will be further explored in the next section.

Proposed Approach

In C programming language, a local stack is used to store local variables and parameters when a function is called. Access to the variables goes through push/pop operations. Energy consumption occurs during the stack process. This usage is further worsening in repeated calls, whereby power drains are inevitable. The excessive energy consumption of stack use can be reduced by code modification to reallocate these local variables to shared memories and register variables, wherever deemed appropriate. In so doing, repetitive accesses to data will lessen stack process considerably, thereby energy consumption is reduced. By the same token, moving local variables to register variables will also accomplish similar energy savings since data access can be done faster and register variables expend less access effort than stacks.

To explore the operating characteristic of parameter and variable allocation at instruction level, a C program is first compiled into assembly code. Each machine instruction is examined to determine the number of clock cycles used [1], depending on types of instruction. For instance, PUSH takes one clock cycle but consumes 3 latency cycles. Measurement is performed in reciprocal throughput (RT) and latency. RT is the average number of core clock cycles per instruction for a series of independent instructions of the same kind in the same thread [1]. Latency of an instruction is the delay that the instruction generates in a dependency chain. The total clock cycles of every instruction used by the entire program can then be converted to energy consumption [3], which is measured as energy per instruction (EPI). The unit of energy from the average EPI is expressed in nano Joule.

In this study, we will compare the amount of energy consumed by an original local variable allocation with that of shared memory and register variable by arranging the same program code in three respective forms, namely, local, shared memory (global), and register variables. Fig. 1 illustrates such an arrangement.

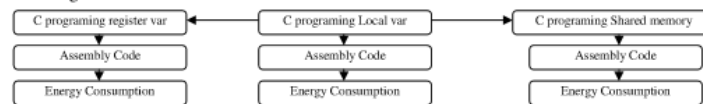


Fig. 1. A comparative energy consumed by local variable, shared memory and register variable.

Two scenarios will be investigated, namely, (1) local variable vs. shared memory or LS, and (2) local variable vs. register variable or LR. A collection of C programs are set up to assist in the analysis. The following case studies will be carried out to exercise both scenarios:

1. Function calls. It is the simplest exercise on parameter allocation, access, and retrieval. Normally, a programmer will use local variables declared in the main function. These variables will subsequently be passed to other functions in the form of parameters. For the first scenario, program modification is done by moving local variables to shared memories, thereby no parameter passing is needed. For the second scenario, the register keyword is simply added to proper local variables.

2. Repeated function calls. The objective is to find code segments that exhibit high energy consumption in a program and behavior of the associated variables/parameters. As such, program improvement can be directed to the right code segments where heavy energy consumptions will be reduced. As a consequence, this case intentionally contrives repeated calls to function for this particular purpose.

3. Function calls to function. This case is intended to investigate the cascading effect of energy consumption consumed by parameter allocation and reference. The complication of such operations, i.e., stack, shared memory, and register variable, at the instruction level are systematically measured and compared.

4. Nested repeated function calls to function. This case culminates all of the above complications to demonstrate as close to actual operation as possible.

Tab.1 illustrates a straightforward comparison of the case studies under both scenarios.

Tab. 1. Example of pseudocode in repeated function calls.

Local variable	Shared memory	Register variable
main() { var A for <i>n</i> times function(A) end for }	var A main for <i>n</i> times function() end for }	main() { register var A for <i>n</i> times function(A) end for }
function(para A)	function()	function(para A)

Experimental and Results

We built a tool called KP program to help analyze the test programs. The tool first read an input C program submitted by the user under the above two scenarios. It located local variables and prompted the user to reallocate or alter them to shared memories or register variables. A lookup table was created by the tool to hold all the data selected by the user for shared variable reallocation or register variable alteration. The tool then compiled both original and modified C programs to produce assembly instructions for determining clock cycles and EPI equivalent. The operating environment was hosted by a laptop computer with Intel Core 2 Duo @ 2.00GHz 65 nm, 3 GB RAM running Windows 7. The tool was coded in C# using Microsoft Visual Studio. All test programs were compiled with MinGW which was a ported GNU compiler collection (GCC).

Code analyses are shown in Fig. 2-3. Fig. 2 depicts KP tool running the original program for repeated function calls (case 2). All variables are locally declared. Fig. 3 shows the reallocation from local variables to shared memories. The numbers of instructions to be executed is less than the original version. Alteration of local variables to register variables is straightforward.

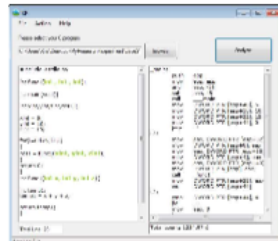


Fig. 2. KP tool running case 2—local variable.



Fig. 3. KP tool running case 2 scenario 1—shared memory.

The rationale behind each case study was to determine the amount of energy consumed by program instructions under different functions. We began with simple function calls (case 1). The tool analyzed and compared clock cycles used, and the energy consumed by each scenario. For example, the original C program contained 30 instructions that utilized 27.31 clock cycles and 300.41 nJ of energy. Under the first scenario (LS), the number of instructions, clock cycles, and energy consumed were 21, 18.98, and 208.78, respectively. Similarly, statistics of the second scenario (LR) came out to be 35, 28.63, and 314.93, respectively. Obviously, shared memories exhibited a sizable savings (-30.50%), while register variables showed a slightly higher consumption (+4.60%) than that of the local variables. The story was different for repeated function calls (case 2), where shared memories continued to saving energy consumption (-35.60%), and register variables gained on the original local variables (-11.90%). As programs became more complicated, savings on energy consumed were even

more noticeable. The function calls to function (case 3) exhibited such benefits. Shared memory savings went from -30.50% to -31.00%, while the numbers on register variable were down from +4.60% to +2.90%. For nested repeated function calls to function (case 4), the numbers were even more interesting. Shared memories showed -34.10%, while register variables were -7.00%. Tab. 2 and 3 summarize all the statistics taking only RT factor into account, while Tab. 4 incorporates additional latency factor. Fig. 4 shows the total clock cycles used in all 4 cases. A similar pattern is obtained by total number of instructions as shown in Fig. 5.

Tab.2. (RT) Instruction clock cycles and number of instructions.

Case	Instruction Clock Cycle			Number of Instruction		
	Local	Register	Shared Memory	Local	Register	Shared Memory
1	27.31	28.63	18.98	30	35	21
2	112.27	98.91	72.27	124	116	84
3	42.97	44.28	29.64	46	52	32
4	190.57	177.16	125.57	204	206	139

Tab. 3. (RT) Energy consumption by allocation scheme (nano Joule).

Case	Energy consumption (nano Joule)					
	Local	Register	Shared memory	Shared Memory VS Local	Register VS Local	Shared Memory VS Register
1	300.41	314.93	208.78	-30.50	+4.60	-33.70
2	1234.97	1088.01	794.97	-35.60	-11.90	-26.93
3	472.67	487.08	326.04	-31.00	+2.90	-33.06
4	2096.27	487.08	1381.27	-34.10	-7.00	-29.12

Tab. 4. (RT+Latency) Energy consumption by allocation scheme (nano Joule).

Case	Energy consumption (nano Joule)					
	Local	Register	Shared memory	Shared Memory VS Local	Register VS Local	Shared Memory VS Register
1	839.41	809.93	538.78	-35.81	-3.51	-33.47
2	3654.97	2375.01	1564.97	-57.18	-35.02	-34.12
3	1352.67	1279.08	854.04	-36.86	-5.44	-33.23
4	6199.27	5446.76	3801.27	-38.68	-12.14	-30.21

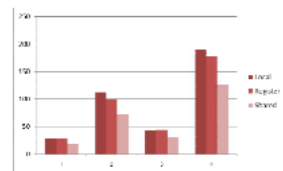


Fig. 4. Total number of clock cycles (Y-axis) for all cases (X-axis).

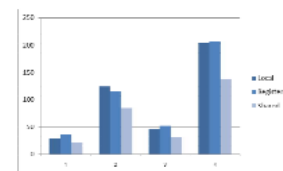


Fig.5. Total number of instructions (Y-axis) for all cases (X-axis).

Discussion

The results obtained from our KP tool did not come as a surprise, as they were well-established programming facts. Our findings merely reinstated local, register, and shared memory variable scoping principles. From our findings, shared memory reduced energy consumption by approximately 30% in simple function calls, and 35% in nested function calls. Register variable, on the other hand, was effective only when repetitive accesses were called for. It was however not as efficient as shared memory. Energy consumption comparison between shared memory and register variable reveals that the former saves approximately 33% and 25% in simple function calls and nested function calls, respectively.

An interesting finding that is not stated in many literatures is the total energy consumption of different variable scoping. We found that local variables consumed the most energy. From the

instruction level, it was apparent that stack process took considerable clock cycles which resulted in high latency and energy consumption. Fig. 6 illustrates the results obtained in Tab. 3, while Fig. 7 shows the results from Tab. 4. The shortfall of shared memory is due to memory access protection as it violates information hiding principle in Software Engineering. The side effect of shared access is what programmers should heed and practice with care.

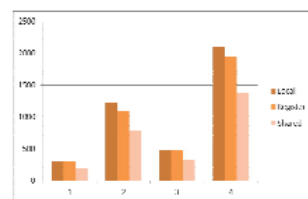


Fig. 6. Comparison of energy consumption (Y) for all cases (X).

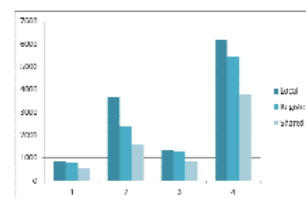


Fig. 7. Comparison of energy consumption (Y) for all cases (X).

Conclusion and Future Work

We investigated energy consumption used by local variable, register variable, and shared memory in C programs. Experiments were conducted in 2 scenarios by means of a KP tool. We found that shared memory significantly saved the most energy consumption over local and register variables. Under repetitive accesses, register variable provided considerable reduction of energy consumption. However, the outcomes were not any surprised. The gains from shared memory could possibly be offset by violation penalty of “good” software engineering practice, e.g., side effects, information hiding, portability, etc. The issue at hand is whether software engineering or energy consumption is crucial to producing theoretically sound or environmentally conscious products.

The KP tool greatly helps identify code fragment to reduce the energy consumption from programming point of view. However, it is currently limited to basic analysis. We plan to improve our tool to be able to analyze complicated C programs in wider research aspects.

References

- [1] Kostas Zotos, Andreas Litke, Er Chatzigeorgiou, Spyros Nikolaidis, George Stephanides, “Energy complexity of software in embedded systems”. ACIT - Automation, Control, and Applications 2005.
- [2] Agner Fog, Instruction Tables Lists of Instruction Latencies, Throughputs and Micro Operation Breakdowns for Intel, AMD and VIA CPUs, Copenhagen University College of Engineering.
- [3] E. Grochowski and M. Annavaram, Energy per instruction trends in Intel microprocessors, Technical report, Microarchitecture Research Lab, Intel Corporation, Santa Clara, CA, Mar 2006.
- [4] S. Eugene, B. Paramvir and Michael J. Sinclair, “Wake on wireless: an event driven energy saving strategy for battery operated devices”, in MobiCom '02 Proceedings of the 8th annual international conference on Mobile computing and networking, 2002, pages 160-171.
- [5] Vivek Tiwari, Sharad Malik, Andrew Wolfe and Mike Tien-Chien Lee, “Instruction Level Power Analysis and Optimization of Software”, Journal of VLSI Signal Processing Systems, 1996, pages 223-238.

- [6] H. Timo, E. Christopher, K. Rüdiger and Wolfgang Schröder-Preikschat, "SEEP: exploiting symbolic execution for energy-aware programming", in HotPower '11 Proceedings of the 4th Workshop on Power-Aware Computing and Systems, No. 4, 2011.
- [7] Mike Tien-Chien Lee and Lee Vivek Tiwaria, "Memory allocation technique for low-energy embedded DSP software", IEEE Symposium on Low Power Electronics, San Diego, CA, Oct 1995.

VITA

Mr. Krisada Samrittianusorn graduated in Bachelors of Multimedia from Swinburne University in 2006. He has interest in green and computing technology, of which made his decision to further his study at Chulalongkorn University mastering in Computer Science and Information Technology. He believes in a world where technology is vital in improving the lives of people and creates a positive society.

