

國立交通大學

電機與控制工程學系

碩士論文

針對嵌入式處理器之原始碼能量最佳化的實驗性分析

An Empirical Analysis of Source-level Energy Optimization for
Embedded Processors

研究生：黃詠文

指導教授：黃育綸 博士

中華民國九十六年九月

針對嵌入式處理器之原始碼能量最佳化的實驗性分析
An Empirical Analysis of Source-level Energy Optimization for
Embedded Processors

研究生：黃詠文

Student : Yung-Wen Huang

指導教授：黃育綸 博士

Advisor : Dr. Yu-Lun Huang

國立交通大學
電機與控制工程學系
碩士論文



Submitted to Department of Electrical and Control Engineering
College of Electrical and Computer Engineering
National Chiao Tung University
in partial Fulfillment of the Requirements
for the Degree of
Master
in

Electrical and Control Engineering

September 2007

Hsinchu, Taiwan, Republic of China

中華民國九十六年九月

針對嵌入式處理器之原始碼能量最佳化的實驗性分析

學生：黃詠文

指導教授：黃育綸 博士

國立交通大學電機與控制工程學系（研究所）碩士班

摘 要

適度地轉換程式原始碼可以有效地減少處理器執行指令的數量、降低指令或資料快取失敗的機會，進而降低存取外部記憶體次數與所需的能量消耗。程式原始碼的轉換概念始於保持轉換前後原始碼的執行結果，但卻可以節省整個程式執行所需耗費的能量。近年來嵌入式系統的市場需求趨勢，強力地引導產業提供更省能的應用與產品，除了電力電子方面的相關研究，在軟體模組能量最佳化方面，由於其不用變動硬體電路的優點，使得許多研究學者投注其相關研究，例如，改變資料結構、迴圈、函式、控制流程與運算子等。然而，在嵌入式處理器日漸普及的趨勢下，我們發現軟體模組能量耗損情形與其所在嵌入式處理器的指令集架構(Instruction Set Architecture, ISA)亦有著密不可分的關係。換言之，程式原始碼的轉換對軟體能量消耗產生的影響會隨著指令集架構的不同，而有所差異。此差異甚至可能改變套用早期程式原始碼轉換所預期的能量耗損，使得轉換後的能量消耗不降反升。因此，在本論文中，我們針對各種軟體能量最佳化轉換方法，加以重新分類，增加了與處理器指令集架構相關的類別，並以 ARM 處理器指令集架構提出兩種新的轉換方法，包括插入冗餘變數、重新排列陣列宣告，嘗試減少存取陣列變數時，計算陣列基底位址時所需的指令個數，藉以降低能量的消耗。此外，為了驗證各種轉換方法所能貢獻的能量最佳化程度以及邊際效應，我們設計了一連串的實驗，以 EMSIM 作為我們的 StrongARM 處理器的能量模擬器，對特定形式的軟體程式碼進行不同的轉換。透過這些實驗，可以取得各種轉換方法所能得到的能量消耗情形，並探討各種轉換所帶來的成本效應，如程式碼大小及程式執行效率的變化等影響。

An Empirical Analysis of Source-level Energy Optimization for Embedded Processors

Student: Yung-Wen Huang

Advisor: Dr. Yu-Lun Huang

Department of Electrical and Control Engineering

National Chiao Tung University

Abstract

Source-level transformations can reduce the number of assembly instructions and the miss rate of instruction or data cache, resulting in an optimization of energy consumption and retaining the same execution results for software modules. Recently, the increasing market demand has become major a driving force for industries to create more energy-efficient applications and products. In addition to power electronics, because of the advantage of hardware circuit remaining unchanged researchers also devote a lot to energy optimization in software modules, such as applying the source-level transformations to data structures, loops, procedures, control structures, operators and so on. We found that energy consumption of software modules is highly related to the instruction set architecture (ISA) of the embedded processors, which means that the expected energy consumption is affected by the ISA of the processors. The result might not be what we expected upon applying the source-level transformations. In this thesis, we re-classify the source-level transformations, and add a new ISA-specific sub-category. Based on ARM ISA, we propose two transformations for energy optimization, called dummy variables insertion and arrays declaration permutation, to reduce the instructions in calculating the base addresses of the arrays. These transformations are verified via a series of experiments based on the EMSIM, the energy simulator of StrongARM. From these experiments, the energy optimization for each transformation can be analyzed and the side effects, such as code size and executing performance, can also be evaluated.

誌 謝

這篇論文能順利完成，首先最要感謝的是我的指導教授黃育綸老師，在研究方面的指導、教誨與鼓勵。在我遇到問題與挫折時，能給予我一盞明燈，指引我明確的道路，以及繼續往前進的動力。感謝您隨和的個性、不給予我們壓力，也讓我們可以在良好的環境下進行研究。感謝您對我匱乏的表達及語文能力，給予的幫助與提醒。而在為人處事方面，也作為我們的楷模，使我們獲益匪淺。在此再度致上萬分的感謝與敬意，謝謝您！除此之外，也感謝我的口試委員，楊武教授、陳右穎教授以及何福軒博士對本篇論文的建議與指教，使我可以了解不足的地方，並對此進行改善。

另外，也感謝「即時嵌入式系統實驗室」的全體成員，雖然相處只有短短一年的時間，但在學業上的切磋與休息時候的抬槓及娛樂，讓我可以活潑、有趣的環境下進行研究，真的很開心能夠認識你們；也感謝608實驗室的老師、學長、同學及學弟們陪伴我一年半的研究生活；也感謝其他交大的師長、同學、朋友們，在學習道路上的指引與陪伴；也感謝大學時代的同窗好友，在研究方面能提供他們的經驗與我分享，在閒暇之餘，陪我聊天、玩遊戲，舒解平日的壓力。

感謝我的家人，父、母以及常常一起出門逛街的大姊跟二姊，從出生開始就一路陪我走來。

最後，向在我人生道路上，一路陪伴我走來的所有人們，獻上最誠摯的祝福與感謝。

Table of Contents

摘要	i
Abstract	ii
誌謝	iii
Table of Contents	iv
List of Tables	vii
List of Figures	viii
Chapter 1 Introduction	1
1.1 Background	1
1.2 Contribution	3
1.3 Synopsis	3
Chapter 2 Related Work	5
2.1 Transformations in Source Code Level	5
2.2 ISA on Energy Consumption	7
2.2.1 Data Processing Instructions of ARM	8
2.2.2 Load and Store Instructions of ARM	9
2.3 APCS on Energy Consumption	10
2.4 Compiler Options that Control Optimization	11
2.5 Evaluation of Energy Consumption	12
2.6 SUIF2 Compiler System	15
Chapter 3 Transformations	17
3.1 Classification / Category	17
3.2 Data Transformations	18
3.2.1 Scratch-pad Array Introduction	18
3.2.2 Local copy of global variable	18
3.2.3 Common Sub-expression Elimination	18
3.2.4 Miscellany	19
3.3 Loop Transformations	19
3.3.1 Loop Fusion	19
3.3.2 Loop Fission	20
3.3.3 Loop Reversal	20
3.3.4 Loop Inversion	20
3.3.5 Loop Interchange	21
3.3.6 Loop Unrolling	21
3.3.7 Loop Unswitching	22
3.3.8 Miscellany	22
3.4 Control Structures and Operators Transformations	23
3.4.1 Conditional Sub-expression Reordering	23

3.4.2 Special Cases Optimization	23
3.4.3 Special Cases Pre-evaluation	24
3.5 Procedural Transformations	24
3.5.1 Procedure Inlining	24
3.5.2 Procedure Integration	24
3.5.3 Procedure Sorting	25
3.5.4 Procedure Cloning	25
3.5.5 Loop Embedding	25
3.5.6 Substitution of a Variable Passed as an Address with a Local Variable	26
3.5.7 Miscellany	26
3.6 ISA-specific Transformations	26
3.6.1 Arrays Declaration Sorting	26
3.6.2 Dummy Variables Insertion	27
3.6.3 Arrays Declaration Permutation	29
3.7 Summary	32
Chapter 4 Experiments	33
4.1 Experimental Framework	33
4.2 Data Transformations	37
4.2.1 Common Sub-expression Elimination	37
4.3 Loop Transformations	38
4.3.1 Loop Fusion	38
4.3.2 Loop Fission	38
4.3.3 Loop Reversal	39
4.3.4 Loop Inversion	40
4.3.5 Loop Interchange	40
4.3.6 Loop Unrolling	41
4.3.7 Loop Unswitching	42
4.4 Control Structures and Operators Transformations	43
4.4.1 Conditional Sub-expression Reordering	43
4.5 Procedural Transformations	43
4.5.1 Procedure Inlining	44
4.5.2 Procedure Integration	45
4.5.3 Loop Embedding	46
4.6 ISA-specific Transformations	46
4.6.1 Arrays Declaration Sorting	46
4.6.2 Dummy Variables Insertion	47
4.6.3 Arrays Declaration Permutation	48
Chapter 5 Results and Analyses	49
5.1 Data Transformations	50

5.1.1 Common Sub-expression Elimination	50
5.2 Loop Transformations	51
5.2.1 Loop Fusion	51
5.2.2 Loop Fission	52
5.2.3 Loop Reversal	53
5.2.4 Loop Inversion	54
5.2.5 Loop Interchange	54
5.2.6 Loop Unrolling	56
5.2.7 Loop Unswitching	59
5.3 Control Structures and Operators Transformations	60
5.3.1 Conditional Sub-expression Reordering	60
5.4 Procedural Transformations	60
5.4.1 Procedure Inlining	61
5.4.2 Procedure Integration	62
5.4.3 Loop Embedding	63
5.5 ISA-specific Transformations	64
5.5.1 Arrays Declaration Sorting	65
5.5.2 Dummy Variables Insertion	67
5.5.3 Arrays Declaration Permutation	69
5.6 Summary	71
Chapter 6 Conclusion and Future work	73
References	74

List of Tables

Table 2-1 ARM Data processing instructions (Seal [27]).....	8
Table 2-2 General and program counter registers [28].....	11
Table 2-3 The optimization levels of gcc 2.95.3 [30].....	12
Table 3-1 Sub-categories of code transformations	17
Table 4-1 The ARM toolchain	34
Table 4-2 The target architecture of our experimental framework.....	37
Table 5-1 The definition of notations	49
Table 5-2 The result of Exp#1.1	50
Table 5-3 The definition of notations used in Section 5.1.1	50
Table 5-4 The result of Exp#2.1	51
Table 5-5 The definition of notations used in Section 5.2.1	52
Table 5-6 The result of Exp#2.2	53
Table 5-7 The result of Exp#2.3	53
Table 5-8 The result of Exp#2.4	54
Table 5-9 The result of Exp#2.5.a	55
Table 5-10 The result of Exp#2.5.b	55
Table 5-11 The result of Exp#2.7.a.....	59
Table 5-12 The result of Exp#2.7.b.....	59
Table 5-13 The result of Exp#3.1.....	60
Table 5-14 The definition of notations used in Section 5.4.....	61
Table 5-15 The result of Exp#4.1	62
Table 5-16 The result of Exp#4.2	62
Table 5-17 The result of Exp#4.3	64
Table 5-18 The definition of notations used in Section 5.5.....	65
Table 5-19 The result of Exp#5.1.a	66
Table 5-20 The result of Exp#5.1.b	66
Table 5-21 The result of Exp#5.2	68
Table 5-22 The result of Exp#5.3	69
Table 5-23 The results after transformation in our experiments.....	71
Table 5-24 The expected results after transformation	72

List of Figures

Figure 2-1 Data processing operands - Immediate (Seal [27]).....	9
Figure 2-2 Modeled embedded system in EMSIM (Tan <i>et al.</i> [32]).....	13
Figure 2-3 Energy analysis framework of EMSIM (Tan <i>et al.</i> [32]).....	14
Figure 2-4 The 32-way set-associative cache in EMSIM.....	14
Figure 2-5 The SUIF system architecture (Aigner <i>et al.</i> [35]).....	15
Figure 2-6 A typical SUIF compiler (Aigner <i>et al.</i> [35]).....	16
Figure 3-1 Some examples of macro definition for procedures (Brandolese <i>et al.</i> [16]).....	24
Figure 3-2 A linked list L used by the algorithm of dummy variables insertion.....	28
Figure 4-1 The execution results of EMSIM.....	34
Figure 4-2 The overall experimental framework.....	35
Figure 4-3 The execution result of the Energy Report program.....	36
Figure 4-4 C source code of common sub-expression elimination for Exp#1.1.....	37
Figure 4-5 C source code of loop fusion for Exp#2.1.....	38
Figure 4-6 C source code of loop fission for Exp#2.2.....	39
Figure 4-7 C source code of loop reversal for Exp#2.3.....	39
Figure 4-8 C source code of loop inversion for Exp#2.4.....	40
Figure 4-9 C source code of loop interchange for Exp#2.5.a.....	40
Figure 4-10 C source code of loop interchange for Exp#2.5.b.....	41
Figure 4-11 C source code of loop unrolling for Exp#2.6.....	42
Figure 4-12 C source code of loop unswitching for Exp#2.7.a.....	42
Figure 4-13 C source code of loop unswitching for Exp#2.7.b.....	42
Figure 4-14 C source code of conditional sub-expression reordering for Exp#3.1.....	43
Figure 4-15 C source code of procedure inlining for Exp#4.1.....	44
Figure 4-16 C source code of procedure integration for Exp#4.2.....	45
Figure 4-17 C source code of loop embedding for Exp#4.3.....	46
Figure 4-18 C source code of arrays declaration sorting for Exp#5.1.a.....	47
Figure 4-19 C source code of arrays declaration sorting for Exp#5.1.b.....	47
Figure 4-20 C source code of dummy variables insertion for Exp#5.2.....	48
Figure 4-21 C source code of arrays declaration permutation for Exp#5.3.....	48
Figure 5-1 The results of the code size in Exp#2.6.....	56
Figure 5-2 The results of the instruction cache misses in Exp#2.6.....	56
Figure 5-3 The results of the CPU cycles in Exp#2.6.....	57
Figure 5-4 The results of the energy consumption in Exp#2.6.....	57
Figure 5-5 The stack content of Exp#5.1.a.....	67
Figure 5-6 The stack content of Exp#5.1.b.....	67
Figure 5-7 The stack content of Exp#5.2.....	68
Figure 5-8 The stack content of Exp#5.3.....	69

Chapter 1

Introduction

Recently, energy consumption in design of embedded systems has become a major issue due to the popularity of portable and mobile products. In software aspect, several approaches are proposed to reduce energy consumption. Transformations in source code level among these approaches are weakly tied to target architecture and are application-independent, so it attracts many researchers' interesting. However, there are many impact factors for transformations and it usually accompanies side effects by using transformations. Hence, it is important to understand these impact factors and evaluate the side effects to get a better trade-off between energy consumption savings, code size and performance.



1.1 Background

With the arrival of mobile generation, there are more and more mobile and portable products of embedded systems on the market. In order to lengthen lifetime of batteries in such products; therefore, energy consumption savings of embedded systems becomes a very important issue. Researches of energy consumption savings are divided into two aspects: hardware and software. Because software programs control the behavior of hardware, energy consumption of the overall embedded systems depend heavily on software design.

It is a critical step to evaluate software energy consumption prior to low energy software design. There are a number of researches about software energy evaluation. Some researches evaluate energy consumption based on physical measurements [1]-[3], and some do it based on simulation [4]-[6].

Low energy software design can be achieved at three levels of abstraction: instruction

level, program or source code level, and algorithm level [7], [8]. There are different research groups devoted to investigation on different levels, respectively. It is natural to hypothesize that the efficiency of analysis, and the amount of energy savings obtainable, are much larger at higher levels [9].


Instruction level approaches focus on better code generation for a program by using energy consumption as the design metric. Such approaches include register allocation to minimize memory access, register relabeling to minimize the switching cost in the instruction register and the decoder [10], and instruction reordering to minimize the switching on the control path [1], etc. Although these approaches can be implemented automatically (the back-end of most compiler can implement many performance-oriented optimizations), the overall energy consumption savings is not remarkable and is strongly tied to the target architecture.

In algorithm level, it can get the highest energy consumption savings by selecting appropriate algorithms in software. For example, Mehta *et al.* [10] evaluated several sorting algorithms, including quicksort, heapsort and bubblesort. They observe that quicksort has less energy consumption than heapsort by using less pointer arithmetic. But algorithm selection is strongly based on programmers' experience and knowledge, it is difficult to implement automatically and needs very large manual effort.

In source code level, it reduces energy consumption by restructuring program code. It also gets balance between efficiency and energy savings. It is weakly target architecture-dependent and is easy to be implemented automatically. There are many approaches in source code level. Tan *et al.* [9] proposed software architectural transformations based on OS-driven multi-process. By analyzing and macro-modelling the energy consumption of various components in an embedded OS [11], they can optimize the energy consumption of embedded software by performing a series of selected software architectural transformations. Peymandoust *et al.* [12] proposed a new methodology based on symbolic

manipulation of polynomials and energy profiling. They use floating-point to fixed-point data conversion and polynomial approximation to achieve a new embedded software optimization methodology. Simunic *et al.* [5] proposed data optimization to match the characteristics of the target architecture with the processed data. They developed a fixed-precision library for processor SA-1100 to replace floating-point arithmetic operations. In [15]-[21], a series of transformations in source code level were presented to reduce energy consumption. This technique is application-independent and can be implemented automatically. The basic principle of transformations is to transform the source code of program such that the transformed result is functionally identical to the original but is much more energy-efficient. In this thesis, we focus on transformations in source code level due to the feature of application-independent and being implemented automatically.

1.2 Contribution



In this thesis, we collect a series of transformations in source code level. Because the impact of ISA on transformations was not considered in previous work, we re-classify the transformations and propose new ones which are ISA-specific on the ARM. We also present an experimental framework and design a number of experiments to verify energy-efficiency of the transformations presented and proposed. The side effects after transformation are also evaluated and discussed.

1.3 Synopsis

The remainder of this thesis is organized as follows. Chapter 2 discusses related work. In Chapter 3, we redefine categories of transformations in source code level and detail the transformations presented. In Chapter 4, an experimental framework is presented and a number of experiments of the transformations are designed to verify their energy-efficiency,

followed by the results and analyses in Chapter 5. Finally the conclusion and future work are given in the last chapter.



Chapter 2

Related Work

As energy consumption in design of embedded systems becomes more and more important, several transformations in source code level [13]-[26] have been proposed to achieve the goal of low energy software design. A number of transformations have been proposed by using different evaluation metrics, such as performance and energy consumption, etc. Besides, researchers don't consider the impact of instruction set architecture (ISA) on energy consumption. In our research, we adopt StrongARM as our target processor and do a number of experiments for the transformations to check if they can be used for energy consumption savings. The impacts of ISA and APCS on energy consumption are discussed, too. Besides, we also find that the optimization levels and options of compiler impact on energy consumption. In order to evaluate energy consumption of software programs, we adopt EMSIM energy simulator [31] as a part of our framework. Finally, the SUIF compiler system [34] which is a compiler infrastructure is discussed. In the system, passes can be developed to do transformations automatically. We use some passes released by other researchers' groups in our experiments.

2.1 Transformations in Source Code Level

According to different requirements such as better performance, smaller code size or lower energy consumption, several transformations in source code level are presented and proposed in [13]-[26]. Russell *et al.* [2] concluded that minimizing software execution time (i.e. improved performance) results in minimized energy consumption. Although it is not always true, we find that improved performance usually accompanies energy consumption

savings.

Optimization is the heart of advanced compiler design. A number of optimizations which may be valuable in improving the performance of the object code produced by a compiler were presented in [13], [14]. Muchnick [13] divided compiler optimizations into two mainly areas: intraprocedural and interprocedural optimizations. Intraprocedural optimizations include redundancy elimination, loop optimizations, procedure optimizations, register allocation, code scheduling, and control-flow and low-level optimizations, etc. Optimization for the memory hierarchy was also presented. Morgan [14] pointed out that the optimizing compiler attempts to use all of the resources of the processor and memory as effectively as possible in executing the application program. Hence, a number of optimizations which are used for transforming the program to get better performance were presented. The optimizations include dominator optimization, interprocedural optimization, dependence optimization, global optimization, instruction scheduling, register allocation, and instruction rescheduling, etc. Compiler optimizations in [13], [14] included many transformations which may reduce energy consumption in source code level.

Brandolese *et al.* [15], [16] presented a methodology and a set of models supporting energy-driven source to source transformations. They grouped source to source transformations into four main categories according to the code structures they operate on: loops, data structures, procedures, and control structures and operators. And a number of transformations in different categories were presented.

Chung *et al.* [17] proposed a new transformation which reduces computational effort by using value profiling and specializing a program for highly expected situations. The goal of this technique is to improve energy consumption and performance by reducing computational effort.

In [18]-[21], a number of transformations which are expected to reduce energy consumption were presented for the purpose of system level power optimization, compiler

optimizations for low power systems, reducing instruction cache energy consumption, and iterative compilation for energy reduction, respectively.

Besides, some transformations presented for different purpose are still useful for energy consumption savings, such as improving data locality with loop transformations [22], augmenting loop tiling with data alignment for improved cache performance [23], optimization of computer programs in C [24], writing efficient C for ARM [25], and transforming and parallelizing ANSI C programs using pattern recognition [26], etc.

2.2 ISA on Energy Consumption

Different ISA of target machine may impact on energy consumption because the source code of program needs to be compiled and assembled to object code according to the instruction set of target machine. The number of instructions generated and which style instructions executed will impact on energy consumption.

In this thesis, we focused on the impact of ARM ISA on energy consumption after transformations, so ARM ISA will be discussed below. The ARM instruction set can be divided into six broad classes of instruction [27]:

- Branch instructions
- Data processing instructions
- Status register transfer instructions
- Load and store instructions
- Coprocessor instructions
- Exception-generating instructions

In our research, we find that data processing, and load and store instructions will impact on energy consumption, so we will detail the two groups later. Based on the following

discussions, we propose new transformations tied to ARM ISA in the ISA-specific transformations section.

2.2.1 Data Processing Instructions of ARM

ARM has 16 data processing instructions as shown in Table 2-1.

Table 2-1 ARM Data processing instructions (Seal [27])

Mnemonic	Opcode	Action
AND	0000	$Rd := Rn \text{ AND shifter_operand}$
EOR	0001	$Rd := Rn \text{ EOR shifter_operand}$
SUB	0010	$Rd := Rn - \text{shifter_operand}$
RSB	0011	$Rd := \text{shifter_operand} - Rn$
ADD	0100	$Rd := Rn + \text{shifter_operand}$
ADC	0101	$Rd := Rn + \text{shifter_operand} + \text{Carry Flag}$
SBC	0110	$Rd := Rn - \text{shifter_operand} - \text{NOT}(\text{Carry Flag})$
RSC	0111	$Rd := \text{shifter_operand} - Rn - \text{NOT}(\text{Carry Flag})$
TST	1000	Update flags after $Rn \text{ AND shifter_operand}$
TEQ	1001	Update flags after $Rn \text{ EOR shifter_operand}$
CMP	1010	Update flags after $Rn - \text{shifter_operand}$
CMN	1011	Update flags after $Rn + \text{shifter_operand}$
ORR	1100	$Rd := Rn \text{ OR shifter_operand}$
MOV	1101	$Rd := \text{shifter_operand}$ (no first operand)
BIC	1110	$Rd := Rn \text{ AND NOT}(\text{shifter_operand})$
MVN	1111	$Rd := \text{NOT shifter_operand}$ (no first operand)

There are 11 addressing modes used to calculate the shifter_operand in an ARM data processing instruction. The impact of the immediate addressing mode on energy consumption is discussed below. As shown in Figure 2-1, this data processing operand provides a constant operand to a data processing instruction. It is encoded in the instruction as an 8-bit *immed_8* and 4-bit *rotate_imm*, so that immediate value is equal to the result of rotating *immed_8* (which will be zero extend to 32-bit firstly) right by twice the value in the *rotate_imm*. Hence,

immediate value must be the value as follows:

- ≤ 255
- a multiple of 4 between 256 and 1023;
- a multiple of 16 between 1024 and 4095
- a multiple of 64 between 4096 and 16383
- ...

If you want to assign a value which is not equal to the above value, you will need more than one instruction to complete your operation.

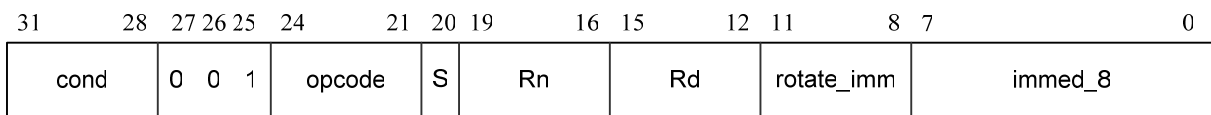


Figure 2-1 Data processing operands - Immediate (Seal [27])

2.2.2 Load and Store Instructions of ARM

Load and store register instructions of load and store instructions are discussed in this section. They use a base register and an offset specified by the instruction. In offset addressing, the memory address is formed by adding or subtracting an offset to or from the base register value. The offset can be either an immediate or the value of an index register. Register-based offsets can also be scaled with shift operations. For the word and unsigned byte instructions, the immediate offset is a 12-bit number. For the halfword and signed byte instructions, it is an 8-bit number. From the above information, we can find that for the word and unsigned byte instructions (or for the halfword and signed byte instructions), if the absolute value of the offset of the memory address from the base address is greater than 4095 (or 255), it can not use an immediate offset and needs another instruction to store the offset to a register; as a result, it needs more than one instruction to load or store the value of a single register from or to memory.

2.3 APCS on Energy Consumption

The APCS (ARM Procedure Call Standard) is a set of rules which regulate and facilitate calls between separately compiled or assembled program fragments [28]. It defines constraints on the use of registers, stack conventions, the format of a stack-based data structure, the passing of machine-level arguments and the return of machine-level results at externally visible procedure calls, and support for the ARM shared library mechanism.

In this section, we discuss that the rules in the APCS that may impact on energy consumption. The ARM has fifteen visible general registers, a program counter register and eight floating-point registers. As shown in Table 2-2, the role of general and program counter registers in the APCS is described. The APCS defines that each contiguous chunk of the stack shall be allocated to activation records in descending address order. At all instants of execution, `sp` shall point to the lowest used address of the most recently allocated activation record. The value of `sl`, `fp` and `sp` shall be multiples of four.

It is noted that the mapping from language-level data types and arguments to APCS words is defined by each language implementation, not by the APCS. Because our research about transformations is focused on C language, C language calling conventions in the APCS are discussed. In an argument list, `char`, `short`, `pointer` and other integral values occupy one word. `Char` and `short` values are widened by the C compiler during argument marshalling. Argument values are marshalled in the order written in the source code of programs. The first four of the remaining argument words are loaded into `a1-a4`, and the remainder are pushed on to the stack in reverse order. A structure is called integer-like if its size is less than or equal to one word, and the offset of each of its addressable sub-fields is zero. An integer-like structured result is returned in `a1`.

Now the APCS is obsolete, and the AAPCS (Procedure Call Standard for the ARM Architecture) should be noted. The AAPCS embodies the fifth major revision of the APCS

and third major revision of the TPCS (Thumb Procedure Call Standard). It forms part of the complete ABI (Application Binary Interface) specification for the ARM architecture [29].

Table 2-2 General and program counter registers [28]

Register	Name	APCS Role
r0	a1	argument 1 / integer result / scratch register
r1	a2	argument 2 / scratch register
r2	a3	argument 3 / scratch register
r3	a4	argument 4 / scratch register
r4	v1	register variable
r5	v2	register variable
r6	v3	register variable
r7	v4	register variable
r8	v5	register variable
r9	sb/v6	static base / register variable
r10	sl/v7	stack limit / stack chunk handle / register variable
r11	fp	frame pointer
r12	ip	scratch register / new-sb in inter-link-unit calls
r13	sp	lower end of current stack frame
r14	lr	link address / scratch register
r15	pc	program counter

2.4 Compiler Options that Control Optimization

In this thesis, gcc 2.95.3 is used as our cross compiler to compile our C source code. Because optimization level and options of compiler impact on energy consumption remarkably, we need to decide what optimization level and options to be used firstly. Optimization levels of gcc 2.95.3 are shown in Table 2-3.

In embedded systems, there are three optimization levels used frequently, -O0, -O2 and -Os. We use -O0 as our optimization level due to stable consideration in embedded systems and clear analysis of the impact of transformations. Besides, we also use -fomit-frame-pointer option to avoid keeping the frame pointer in a register for procedures that don't need one and

avoid the instructions to save, set up and restore frame pointers; as a result, it makes an extra register available to be used. It also makes debugging impossible on ARM.

Table 2-3 The optimization levels of gcc 2.95.3 [30]

Optimization Level	Description
-O0 (default)	This is the default. Do not optimize. In this level, the compiler's goal is to reduce the cost of compilation and to make debugging produce the expected results. The compiler only allocates variables declared register in registers.
-O1 (-O)	Optimize. The compiler tries to reduce code size and execution time.
-O2	Optimize even more. GCC performs nearly all supported optimizations that do not involve a space-speed trade-off. It turns on all optional optimizations except for loop unrolling, function inlining, and strict aliasing optimizations. It also turns on the 'f' option on all machine.
-O3	Optimize yet more. It turns on all optimizations specified by '-O2' and also turns on the 'inline-functions' option
-Os	Optimize for size.

2.5 Evaluation of Energy Consumption

In order to analyze the impact of transformations on energy consumption, we need to find a way to evaluate software energy consumption. Tan *et al.* [31] presented an energy simulation framework that can be used to analyze the energy consumption characteristics of an embedded system featuring the embedded Linux OS running on the StrongARM processor.

As shown in Figure 2-2, the simulator includes the following component:

- 1) a model for the StrongARM SA-1100 core, consisting of an instruction set simulator (ISS), simulation models for the instruction cache and data cache and a memory management unit (MMU);
- 2) a simulation model for 32 MB of system memory;
- 3) a simulation model for an interrupt controller;
- 4) simulation models for two timers;

5) simulation models for two UARTs conforming to the Intel 8250 series.

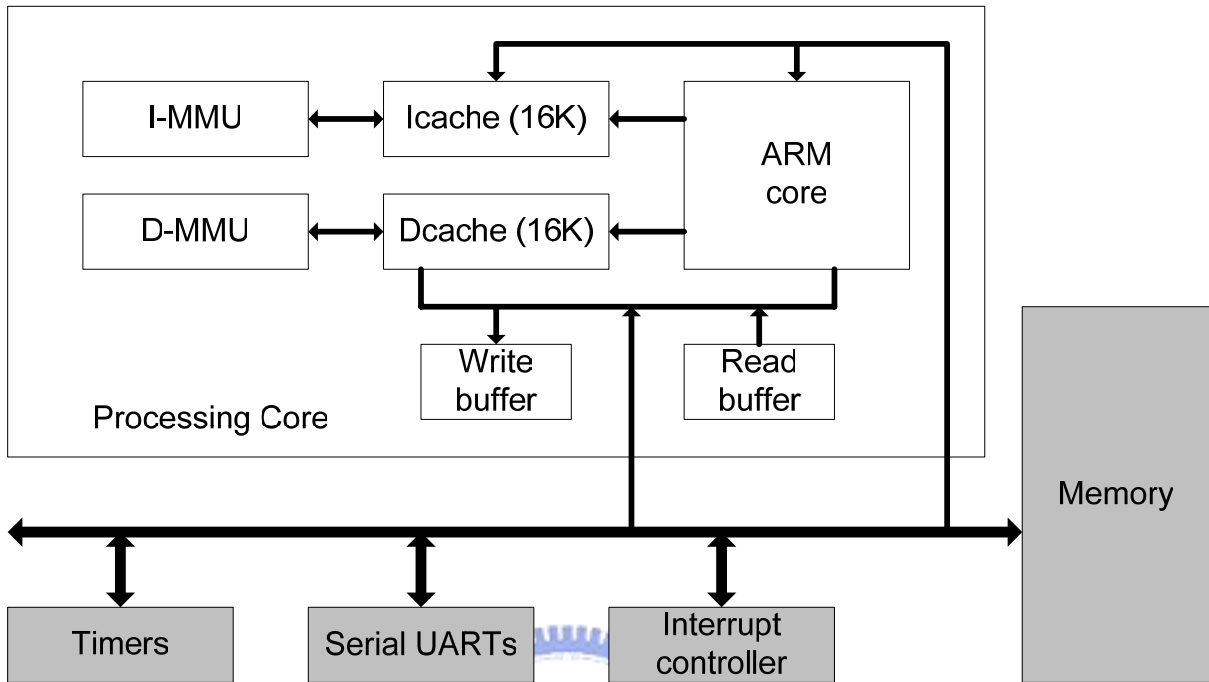


Figure 2-2 Modeled embedded system in EMSIM (Tan *et al.* [32])

The simulation models are shown on the right half of Figure 2-3 and the sequence of steps involved in using the simulation framework is shown on the left. The energy accounting mechanism of EMSIM is task-based. And a function energy stack for each task is used for evaluating the energy consumption of every function in the task. From energy profiling report, we can get information about the number of invoked times, CPU cycles consumed and energy consumption of every function.

The SA-1100 microprocessor is a general-purpose, 32-bit RISC microprocessor with a 16 Kbytes instruction cache, an 8 Kbytes write-back data cache, a minicache, a write buffer, a read buffer, and a memory management unit (MMU) combined in a single chip [33]. Besides, it is software compatible with the ARM V4 architecture processor family. In EMSIM, the 8 Kbytes write-back data cache is replaced by 16 Kbytes one and it doesn't simulate a minicache. As shown in Figure 2-4, the size of the cache line (block) is 32 bytes and the

caches are 32-way set-associative caches. Replacement policy is round robin within a set.

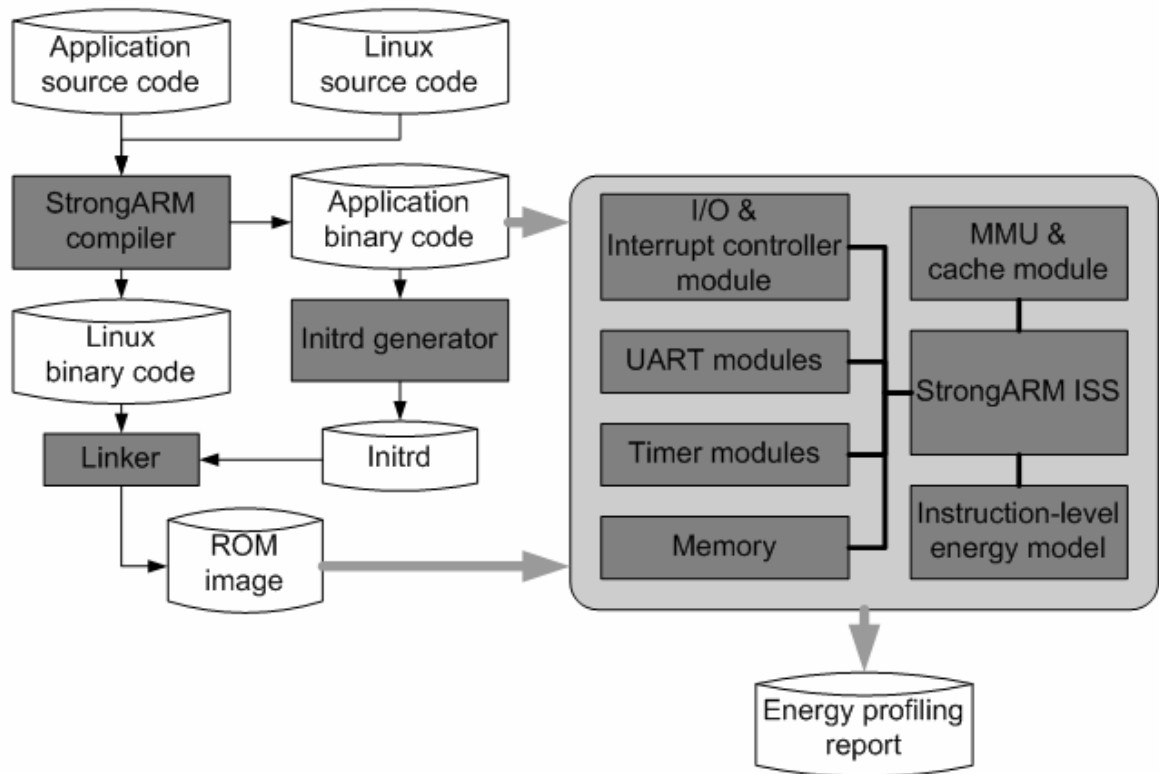


Figure 2-3 Energy analysis framework of EMSIM (Tan *et al.* [32])

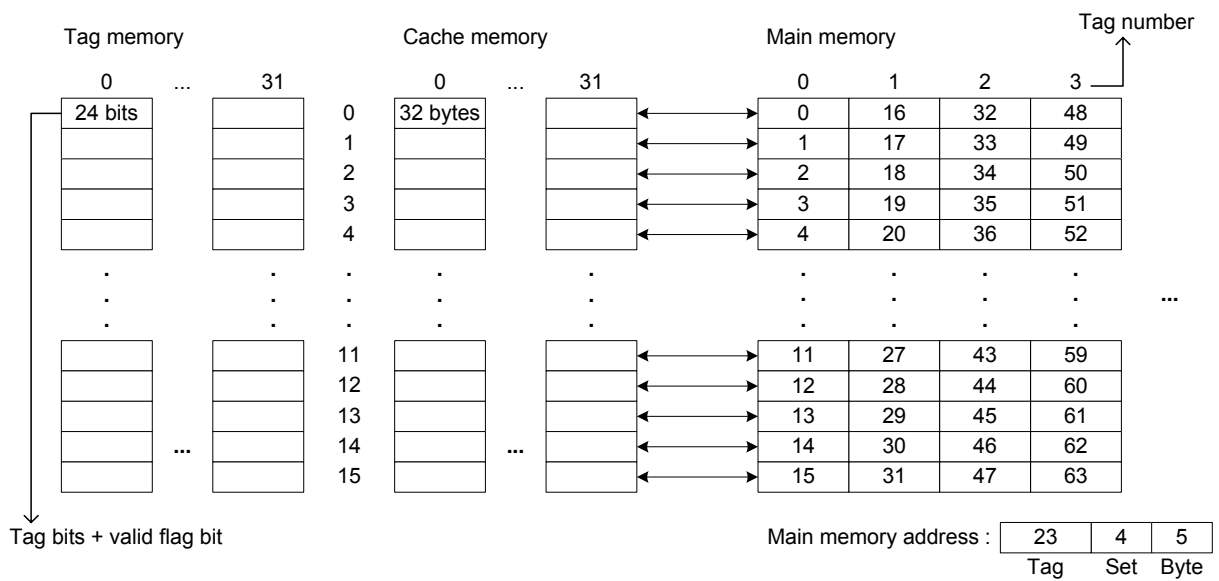


Figure 2-4 The 32-way set-associative cache in EMSIM

2.6 SUIF2 Compiler System

The SUIF (Stanford University Intermediate Format) system [34] was developed by Stanford Compiler Group. It is a free compiler infrastructure designed to support collaborative research in optimizing and parallelizing compilers, based upon a program representation, SUIF. It maximizes code reuse by providing useful abstractions and frameworks for developing new compiler passes and by providing an environment that allows compiler passes to inter-operate easily. Now the SUIF group has moved its effort on from SUIF1 to SUIF2.

It also supports some useful tools, such as front ends, converters from SUIF1 to SUIF2 and vice versa, and converters from SUIF2 back to C, etc. Hence, we can write our SUIF compiler to do operations between SUIF intermediate representations (IRs).

Figure 2-5 shows the SUIF system architecture. The components of the architecture are described as follows.

- 1) Kernel provides all basic functionality of the SUIF system.
- 2) Modules can be one of two kinds: a set of nodes in the intermediate representation and a program pass.
- 3) Suifdriver provides execution control over modules.

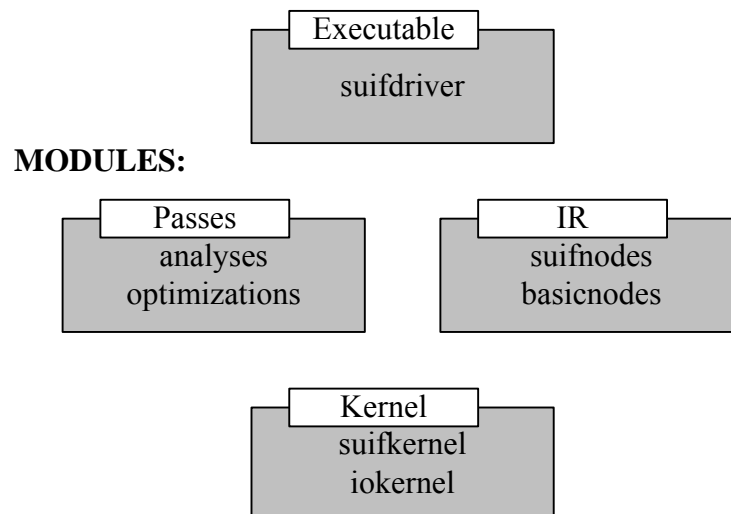


Figure 2-5 The SUIF system architecture (Aigner *et al.* [35])

Passes are the mainly part of a SUIF compiler. It typically performs a single analysis or transformation and then writes the results out to a file. To create a compiler or a standalone pass, the user needs to write a “main” program that creates the SuifEnv, imports the relevant modules, loads a SUIF program and applies a series of transformations on the program and eventually writes out the information, as show in Figure 2-6.

Some passes which do transformations were implemented and released in [36], [37]. We would like to thank for their release, so we can use the passes to do some transformations for our experiments.

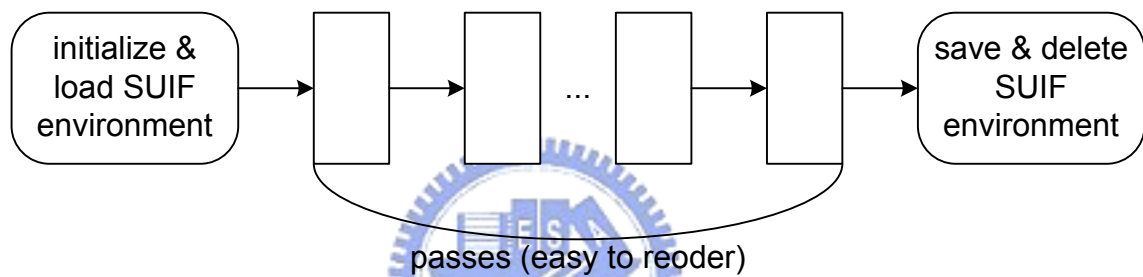


Figure 2-6 A typical SUIF compiler (Aigner *et al.* [35])

Chapter 3

Transformations

A series of transformations operating in source code level were presented in [13]-[25]. In the first section, we firstly explain how we classify transformations, followed by the detail of the transformations in different categories in section 3.2-3.6. Finally the summary of the transformations is given in the last section.

3.1 Classification / Category

According to transformations operating on data or code segments, we firstly divide transformations into two main categories, data and code transformations. In [16], code transformations were grouped into three sub-categories according to the code structures they operate on, including loop, procedural, and control structures and operators transformations. But they don't consider the influence of ISA on energy consumption.

Table 3-1 Sub-categories of code transformations

Sub-category of code transformations	Description
Loop transformations	Modify either the body or control structure of the loop
Control structures and operators transformations	Change either specific control structures or operators
Procedural transformations	Modify the interface, declaration or body of procedures
ISA-specific transformations	Transformations are impacted by ISA

In our research, we find that some code transformations are strongly tied to ISA of target machine. Therefore, our code transformations will include four sub-categories: loop, control

structures and operators, procedural, and ISA-specific transformations. Sub-categories of code transformations are described in Table 3-1.

3.2 Data Transformations

In this section, we present a series of transformations used in modifying data segment of source code. These transformations may result in reduced data cache misses and memory access, etc., and then energy consumption savings is expected.

3.2.1 Scratch-pad Array Introduction

Allocating a smaller array is used in storing the most frequently accessed elements of the larger array [16]. It is expected that spatial locality is improved contributing to reduced data cache misses. It is noted that the increased instructions which are used in refreshing the elements of arrays may reduce performance and increase code size (i.e. instruction cache misses). As a result, it may not reduce energy consumption.

3.2.2 Local copy of global variable

In the procedure which needs to operate on global variables, we can declare local variables and assign the value of global variables to them before the procedure invoked [15]. We then refresh global variables after leaving the procedure. In such a way, this transformation can increase the possibility for compiler to store variables in registers instead of memory (i.e. it reduces data cache misses). But it has the same side effects as above transformation, it may not reduce energy consumption.

3.2.3 Common Sub-expression Elimination

An existence of an expression in a program is a common sub-expression if there is

another existence of the expression whose evaluation always precedes this one in execution order and if the operands of the expression remain unchanged between the two evaluations [13]. Common sub-expression elimination is a transformation which stores the same computation results of common sub-expressions into variables and assigns the value of variables to replace common sub-expressions.

It is noted that this transformations may not always be valuable, because it may be less energy consumption to recompute, rather than allocate another register (or memory) to hold the value. As a result, it does not always reduce energy consumption.

3.2.4 Miscellany

In [15], [16], there are still a number of data transformations presented. Scalarization of array elements introduces temporary variables as a substitute of the most frequently accessed elements of an array. Multiple indirection elimination finds common chains of indirections and uses a temporary variable to store the address. At present, researches about code transformations are still continued proceeding.

3.3 Loop Transformations

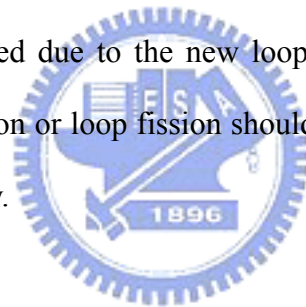
Loop transformations operate on the statements which comprise a loop (i.e. these transformations modify either the body or control structure of the loop). Because a large percentage of the execution time of programs is spent in loops, these transformations can have a very remarkable impact on energy consumption. Hence, there are a number of researches and approaches based on loop transformations because of the importance of loop transformations.

3.3.1 Loop Fusion

This transformation combines one or more loops with the same bounds into a single loop [21]. It reduces loop overhead; as a result, the number of the instructions executed is reduced. Besides, it can also be used for improving data cache locality by bringing the statements that access the same set of data to the same loop [20]. But it is noted that if the increased loop body becomes larger than instruction cache, it will increase instruction cache misses; as a result, energy consumption will be increased.

3.3.2 Loop Fission

Loop fission does the opposite operation to loop fusion [21]. The goal of this transformation is to break down larger loop body into smaller ones to reduce the size of loop body to fit into instruction cache, and then it reduces instruction cache misses. It is noted that computation energy is increased due to the new loop overheads. Therefore, we need to be careful to decide that loop fusion or loop fission should be applied to loops in order to reduce energy consumption effectively.



3.3.3 Loop Reversal

This transformation reverses the order in which a specific loop's iterations are performed [13]. In some loops of which loop body exists dependence, loop reversal may eliminate the dependence; as a result, it allows other transformations to be applied. Besides, a special case of loop reversal which is useful in ARM architecture was presented in [25]. To apply loop reversal transformation makes an incrementing loop to a decrementing loop which becomes a count-down-to-zero loop. It causes the original ADD/CMP instruction pair to be replaced by a single SUBS instruction; because of this, it saves compares in critical loops, leading to reduced code size, increased performance and reduced energy consumption.

3.3.4 Loop Inversion

Loop inversion transforms a *while* loop to a *repeat* loop (i.e. it moves the loop conditional test from before the loop body to after it) [13]. It results in only one branch instruction needed to be executed to leave the loop, rather than one needed to return to the beginning and another needed to leave after the loop conditional test at beginning. Hence, it is expected that energy consumption is reduced due to the reduced number of the instructions executed. It is noted that this transformation is only safe when the loop body is executed at least once.

3.3.5 Loop Interchange

This transformation reverses the order of two adjacent loops in a loop nest to change the access paths to arrays [14]. It improves the chances that consecutive references are in the same cache line, leading to reduced data cache misses. Hence, reduced energy consumption can be expected.



3.3.6 Loop Unrolling

Loop unrolling replaces the body of a loop by U (the unrolling factor) times copies of the body and modifies the iteration step from 1 to U [13]. The original loop is called the rolled loop. Loop unrolling reduces the overhead of a loop by performing less compare and branch instructions (i.e. better performance) and may improve the effectiveness of other transformations, such as common-sub-expression elimination and software pipelining, etc. It also allows the compiler to get a better register usage of the larger loop body.

On the other hand, the unrolled loop is larger than the rolled loop, so it increases code size and may impact the effectiveness of the instruction cache, leading to increased instruction cache misses. So deciding which loops to unroll and by what unrolling factors is very important.

Besides, there is another form of loop unrolling that applies to the loop which is not a

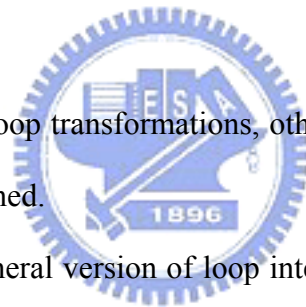
counting loop [14]. It can unroll the loop and leave the termination conditions in place. This technique has benefits when dealing with a *while* loop in which later transformations can be used.

3.3.7 Loop Unswitching

This transformation moves loop-invariant conditional branches to the outside of loops [13]. It reduces the number of the instructions executed due to the reduced number of codes executed in the loop body. If the conditional only has *if* part, loop unswitching has little impact on code size. But if the conditional has *else* parts, it will need to copy the loop into every *else* parts of conditional. Hence, it increases code size and instruction cache misses.

3.3.8 Miscellany

In addition to the above loop transformations, other energy-efficiency strategies on loop transformations can be envisioned.



Loop permutation is a general version of loop interchange [13]. It allows more than two loops to be reordered to reduce data cache misses; as a result, it is expected to reduce energy consumption.

Loop tiling achieves the goal of reduction of capacity and conflict misses which are resulted from cache size limitations [21]. It improves cache performance by dividing the loop iteration space into smaller tiles. This also results in a logical division of arrays into tiles, so it may increase reuse of array elements within each tile. By the correct selection of tile sizes, conflict misses which occur when several data elements compete for the same cache line can be eliminated. But it also increases the number of instructions executed and code size because of the increased nesting loops.

Software pipelining can improve the execution performance of loops [16]. It eliminates the dependences between adjacent statements by breaking the operations of single loop

iteration into S stages, and arranges the code in such a way that stage 1 is executed on the instructions originally belonging to iteration i, stage 2 on those of iteration i-1, etc. It makes pipeline performance better through pipeline stalls reduction. Hence, CPU cycles consumed are expected to reduce. But it may increase the number of instructions executed due to the calculation of iteration i. It also increases code size because startup code is generated before the loop to initialize and cleanup code is generated after the loop to finish operations.

3.4 Control Structures and Operators Transformations

In this section, we present a series of transformations which modify control structures and operators of source code to reduce energy consumption.

3.4.1 Conditional Sub-expression Reordering

It is possible to reorder sub-expressions in conditional to reduce energy consumption [16]. In OR conditions, we can sort the sub-expressions in which the possibility of being true in front of others. In AND conditions, we can sort the sub-expressions in which the possibility of being false in front of others. By using this transformation, it reduces the number of the instructions executed; as a result, it reduces energy consumption. It has no side effects.

3.4.2 Special Cases Optimization

Brandolese *et al.* [16] presented special cases optimization transformation which replaces calls to generic library or user-defined functions with optimized ones. For example, someone needs to call mathematic functions of which arguments need floating-point variables, but he only wants to do operations for integers. Hence, he can re-write optimized ones for integers to reduce energy consumption. This transformation is only a suggestion, and it can not be implemented by an automation tool.

3.4.3 Special Cases Pre-evaluation

Some functions would return a known value when a special value for an argument is passed. So we could avoid real calls to the functions by defining suitable macros testing for the special cases [16]. Hence, it may reduce real function calls (i.e. the number of the instructions executed) but increases code size. Figure 3-1 shows some examples.

Macro name	Macro definition
acos(x)	$((x == -1) ? 3.141592653589793238462643383 : \text{acos}(x))$
asin(x)	$((x == 0) ? 0 : \text{asin}(x))$
pow(x, y)	$((y == 1) ? x : \text{pow}(x, y))$
sqrt(x)	$((x == 0) ? 0 : (x == 1) ? x : \text{sqrt}(x))$
fabs(x)	$((x >= 0) ? x : -x)$

Figure 3-1 Some examples of macro definition for procedures (Brandolese *et al.* [16])

3.5 Procedural Transformations

Procedural transformations are used for modifying the interface, declaration or body of procedures. There are also a number of researches in this sub-category for the purpose of performance improved and energy consumption savings.

3.5.1 Procedure Inlining

This transformation is supported by many compilers. It replaces the invoked procedure with the procedural body [16]; as a result, it increases the spatial locality and decreases the number of procedure invoked. But it increases the code size which will result in increased instruction cache misses.

3.5.2 Procedure Integration

It has almost the same behavior as procedure inlining [13], but procedure inlining does not consider the call site. This transformation can differentiate among call sites which invoke

the same procedure and decide which call site is need to do procedure integration or only invokes the original procedure. As a result, it may get a better trade-off between code size and energy consumption than procedure inlining.

3.5.3 Procedure Sorting

This transformation is the easiest instruction cache optimization approach to implement. It sorts the statically linked procedures according to the call graph and frequency of use [13]. This transformation has two advantages. Firstly, it places procedures near their callers in virtual memory so as to reduce paging traffic. Secondly, it places frequently used and related procedures so they have less possibility to collide with each other in the instruction cache. To implement this transformation, we only need to reorder the procedural declarations.

3.5.4 Procedure Cloning

This transformation is based on procedural parameters which are constant at one or more call sites. For every call site that calls the same procedure and passes the same constant values of parameters, we clone a copy of the procedure and rename its procedural name [13]. The new version of the procedure has reduced parameters and in the body of which constant parameters are replaced by constant values; as a result, it allows compilers to do advanced optimization.

3.5.5 Loop Embedding

Loop embedding is an interprocedural transformation which moves the loop from the outside of a procedure to the body of the procedure [26]; as a result, it reduces the overhead of the procedure call. The original procedure is needed to be reserved if it is called from more than one call site.

3.5.6 Substitution of a Variable Passed as an Address with a Local Variable

This transformation replaces a procedural argument passed as an address with a local copy of variable [15]. In optimization level of compilers, compilers tend to allocate local variables in registers instead of memory. Hence, it reduces the number of memory access and the data cache misses. But it increases the number of codes which assigns and restores values, it will increase the number of the instructions executed; as a result, it may not reduce energy consumption.

3.5.7 Miscellany

Other transformations which operate on procedure include soft inlining which replaces calls and returns with jumps [16], and procedure splitting which divides each procedure into a primary and a secondary component [13], etc.

3.6 ISA-specific Transformations

Some transformations are dependent to what ISA you operate on. In this section, we present one transformation which is not independent of ISA of target machine. We also propose two transformations which are specific to ARM ISA, including dummy variables insertion and arrays declaration permutation transformations. It is noted that the proposed ones are also strongly tied to the strategies of calculating base addresses of compilers.

3.6.1 Arrays Declaration Sorting

This transformation is to modify the order of local arrays declaration, so that the most frequently accessed array is allocated on the top of the stack; in such a way, the memory locations frequently accessed by exploiting direct access mode [15]. It is less energy expensive in this access mode. When using this transformation, you need to know the stack

allocation strategies of local arrays implemented by compilers.

3.6.2 Dummy Variables Insertion

This transformation proposed is based on the feature of ARM ISA. When elements of arrays are accessed, it is necessary to calculate the base addresses of the arrays firstly. This transformation tries to reduce the number of the instructions executed for calculating the base addresses by inserting dummy variables which are declared as volatile ones between the arrays. In such a way, the offsets of the base addresses of arrays from the stack are changed, so that it is possible to use one instruction to get the base addresses of arrays. Because the order of array declarations is changed and the size of stack allocation is increased, it might increase data cache misses and page fault slightly. It is expected that code size and energy consumption will be reduced because of the reduced number of the instructions executed. It is noted that we need to take care of checking if stack overflow will happen after dummy variables insertion.

In this thesis, we follow the pseudocode writing rules in [38] to write our algorithms. We design an algorithm for dummy variables insertion transformation, and we also take some assumptions as follows.

- 1) Offset is equal to or less than 2^{26} (for the procedure DUMMY-VARIABLE-SIZE(offset) to operate correctly).
- 2) We suppose that compilers only use 'add' instruction to calculate the base addresses of arrays, and the immediate value of the instruction must not be negative value.
- 3) In order to simplify our algorithm, we also suppose that initial offset passed to the procedure DUMMY-VARIABLES-INSERT(L , $init_offset$) is equal to or less than 1024 (because the initial offset is a multiple of 4, we don't need to insert dummy variable in the above situation). It is large enough in general cases.

List L passed to the procedure `DUMMY-VARIABLES-INSERT(L , $init_offset$)` is used in storing attributes of local array variables by the reverse order of declaration (i.e. the first element of list L stores the attributes of the rightmost array variable, and the last element of list L stores those of the leftmost one). As shown in Figure 3-2, each element of a linked list L is an object with a string field: var_name , two integer fields: $sizeof_type$ and $no_elements$, and a pointer field: $next$. Given an element x in the list, var_name stores the name of the array variable, $sizeof_type$ stores the size of the element of that, $no_elements$ stores the number of the elements stored in this array, and $next[x]$ points to its successor in the linked list. Besides, an attribute $head[L]$ points to the first element of the list and an attribute $length[L]$ stores the number of elements of the list.

Because compilers will allocate memory space on the top of stack, when the procedure invokes other procedures of which the numbers of the arguments are greater than 4, the initial offset from the top of stack may not be zero. We pass the integer $init_offset$ to the procedure `DUMMY-VARIABLES-INSERT(L , $init_offset$)` to point out this offset.

And the procedure `DUMMY-VARIABLE-SIZE($offset$)` is used in calculating the size of dummy array variable which needs to insert between arrays.

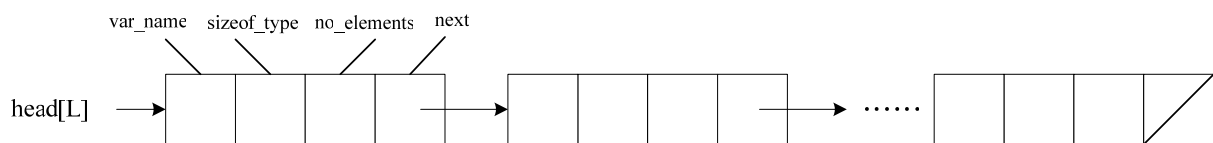


Figure 3-2 A linked list L used by the algorithm of dummy variables insertion

The following is the algorithm of dummy variables insertion.

Algorithm 3.1 Dummy Variables Insertion

`DUMMY-VARIABLES-INSERT(L , $init_offset$)`

- 1 $no_dummy_vars \leftarrow 0$
- 2 $offset \leftarrow init_offset$

```

3  sizeof_type[y] ← 4
4  x ← head[L]
5  n ← length[L] - 1
6  for i ← 1 to n
7      do sizeof_array ← sizeof_type[x] × no_elements[x]
8          remainder ← sizeof_array mod 4
9          if remainder ≠ 0
10             then padding ← 4 - remainder
11             else padding ← 0
12         offset ← offset + sizeof_array + padding
13         dummy_size ← DUMMY-VARIABLE-SIZE(offset)
14         if dummy_size ≠ 0
15             then no_dummy_vars ← no_dummy_vars + 1
16                 offset ← offset + dummy_size
17                 var_name[y] ← “dummy” + to_string(no_dummy_vars)
18                 no_elements[y] ← dummy_size / 4
19                 next[y] ← next[x]
20                 next[x] ← y
21                 x ← next[y]
22         else x ← next[x]

```

DUMMY-VARIABLE-SIZE(offset)

```

1  if offset < 256
2      then return 0
3  bound ← 1024
4  mul ← 4
5  while TRUE
6      do if offset < bound
7          then return mul - (offset mod mul)
8          else bound ← bound × 4
9          mul ← mul × 4

```

3.6.3 Arrays Declaration Permutation

This transformation uses arrays declaration permutation instead of dummy variables insertion to try to reduce the number of the instructions executed for calculating the base addresses of arrays. It modifies the order of local arrays declaration to change the offsets of

the base addresses of arrays from the stack.

We design an algorithm for arrays declaration permutation transformation, and we take the same assumptions used in Algorithm 3.1 (but the procedural names in the assumptions must be replaced suitably). In addition, in order to simplify this algorithm, we use the procedure ENERGY-COST(offset) to get energy cost for calculating base address of an array; in the procedure, if we only need one instruction for calculating base addresses it will return 1 and if we take more than one instruction for calculating base addresses it will return 2 simply.

We refer to the recursive algorithm of permutation algorithms in [39] to design the procedure PERMUTATION(V, k, A, init_offset). In Algorithm 3.2, we use *G_varname* to indicate that this variable is a global variable.

Comparing with Section 3.6.2, list L used in this section is similar except that the object of the element of a linked list L has an extra integer field: *no_cal_base_address*. It stores the number of calculating base address of an array.

The following is the algorithm of arrays declaration permutation.

Algorithm 3.2 Arrays Declaration Permutation

ARRAYS-DECLARATION-PERMUTATION(L, init_offset)

```
1  G_min_t_energy_cost ← ∞
2  G_n ← length[L]
3  G_level ← -1
4  x ← head[L]
5  for i ← 1 to G_n
6      do V[i] ← 0
7          A[i] ← x
8          x ← next[x]
9  PERMUTATION(V, 1, A, init_offset)
10 head[L] ← A[G_min_V[1]]
11 for i ← 1 to G_n - 1
12     do x ← A[G_min_V[i]]
13         y ← A[G_min_V[i+1]]
14         next[x] ← y
15 x ← A[G_min_V[G_n]]
```


16 next[x] ← NIL

PERMUTATION(V, k, A, init_offset)

```
1  G_level ← G_level + 1
2  V[k] ← G_level
3  if G_level = G_n
4    then t_energy_cost ← TOTAL-ENERGY-COST(V, A, init_offset)
5         if t_energy_cost < G_min_t_energy_cost
6             then G_min_t_energy_cost ← t_energy_cost
7                 for i ← 1 to G_n
8                     do G_min_V[i] ← V[i]
9  else for i ← 1 to G_n
10     do if V[i]=0
11         then PERMUTATION(V, i, A, init_offset)
12 G_level ← G_level - 1
13 V[k] ← 0
```

TOTAL-ENERGY-COST(V, A, init_offset)

```
1  offset ← init_offset
2  t_energy_cost ← 0
3  for i ← 1 to n
4    do x ← A[V[i]]
5        t_energy_cost ← t_energy_cost + no_cal_base_address[x] ×
ENERGY-COST(offset)
6        sizeof_array ← sizeof_type[x] × no_elements[x]
7        remainder ← sizeof_array mod 4
8        if remainder ≠ 0
9            then padding ← 4 - remainder
10           else padding ← 0
11        offset ← offset + sizeof_array + padding
```

ENERGY-COST(offset)

```
1  if offset < 256
2    then return 1
3  bound ← 1024
4  mul ← 4
5  while TRUE
6    do if offset < bound
7        then if (offset mod mul) ≠ 0
```



```
8           then return 2
9           else return 1
10         else bound ← bound × 4
11         mul ← mul × 4
```

3.7 Summary

Some transformations are strongly tied to which optimization level of the compiler used. For example, because the gcc compiler only allocates variables declared register in registers. When optimization is not enabled [30], transformations such as local copy of global variable and substitution of a variable passed as an address with a local variable are useless in this case. In addition, some transformations have better energy-efficiency when optimization is enabled.

Besides, some transformations may have no impact on energy consumption, but they can reduce dependency for other transformations to be applied. Or after doing some transformations, it is possible to increase the energy-efficiency by using other transformations.

Although a number of transformations can reduce energy consumption remarkably, it is not easy to find which transformations should be used, in which order to apply, and to which code sections [21]. This long standing open problem is called the phase-order problem. In this thesis, we mainly focus on applying one transformation to the source code every time and evaluate the impact on energy consumption on our target architecture.

Chapter 4

Experiments

Hill *et al.* [40] subdivided set-associative misses into three categories: (set-)conflict misses (due to too many active blocks mapping to a fraction of the sets), *capacity* misses (due to fixed cache size), and *compulsory* misses (those necessary misses caused by the first data access). In Chapter 4 and 5, we use their categories of set-associative misses to explain the experiments designed and the impact of different categories of cache misses on energy consumption respectively.

In Chapter 3, we collect a series of transformations in source code level. The efficiency of the transformations is needed to be evaluated and verified on our target architecture by doing experiments. In this chapter, we design an experimental framework to profile the experimental results in the first section. In the last five sections, the experiments of different categories are designed and completed.

4.1 Experimental Framework

EMSIM 2.0 energy simulator of StrongARM is adopted as a part of our experimental framework to get the energy information of the experiments.

In addition to energy consumption, we need other information to analyze and evaluate side effects when every time simulation functions run. Because the main impact factors on energy consumption include CPU cycles consumed, and the number of instruction and data cache misses, we modify EMSIM energy simulator to get such information. The addresses of instruction and data cache miss are outputted to verify the correctness of cache misses. In addition, in order to get more accurate information about cache misses, we detect if

simulation function will be executed. Before being executed, we flush instruction and data caches. The execution results of EMSIM before and after modified are shown in Figure 4-1. And the information about function code size is got by using arm-linux-objdump program of the GNU Binutils which are a collection of binary tools.

Execution result before modified	Execution result after modified
<pre>... energy_report sim_func 20855.388350 energy_report sim_func 18225.000000 energy_report sim_func 18225.000000 energy_report sim_func 18225.000000 ...</pre>	<pre>... Flush caches reading: the address of instruction cache miss: 0x81ec writing: the address of data cache miss: 0xbffff18 reading: the address of instruction cache miss: 0x8200 reading: the address of instruction cache miss: 0x8220 writing: the address of data cache miss: 0xbffffbc ... reading: the address of instruction cache miss: 0x82a0 energy_report sim_func 21199.077670 11492 cycles, 7 Icache_misses, 80 Dcache_misses Flush caches reading: the address of instruction cache miss: 0x81ec ... reading: the address of instruction cache miss: 0x82a0 energy_report sim_func 21199.077670 11492 cycles, 7 Icache_misses, 80 Dcache_misses ...</pre>

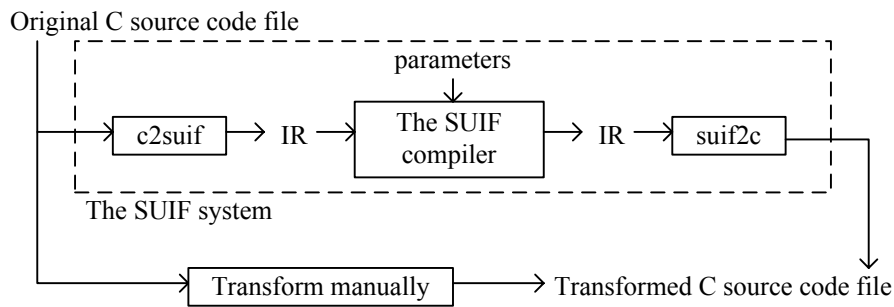
Figure 4-1 The execution results of EMSIM

Since the EMSIM simulation framework is about running the Linux OS in a StrongARM simulator, several Linux and StrongARM related components are needed [31], including Linux OS kernel and the ARM toolchain. Linux 2.4.18 and patch-2.4.18-rmk3 are used in building our Linux kernel. Besides, the ARM toolchain which we build are listed in Table 4-1.

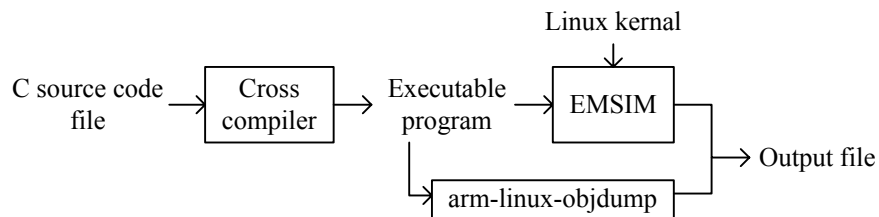
Table 4-1 The ARM toolchain

The ARM toolchain	binutils 2.11
	gcc 2.95.3
	glibc 2.2.3
	glibc-linuxthreads 2.2.3

Step one :



Step two :



Executing shell script “run_batch” to repeat the above process for every C source code file in the same directory

Step three :



Figure 4-2 The overall experimental framework

Our experiments for every transformation involve three steps, comprised of generating C source code files in the same directory, generating output files which record execution results of EMSIM and arm-linux-objdump, and executing “Energy Report” program to profile energy consumption and side effects by parsing output files. The overall experimental framework is shown in Figure 4-2. In step one, we firstly design an original C source code file and use two ways to generate transformed ones, including using SUIF passes which do transformations, and transforming manually. In step two, in order to get the simulation results for every source code file, we write a shell script, namely “run_batch”, to repeat the process of generating

executable program, running simulation and writing out information to output file. In the last step, an “Energy Report” program is designed to parse output files, and to calculate and show the results in GUI, as shown in Figure 4-3.

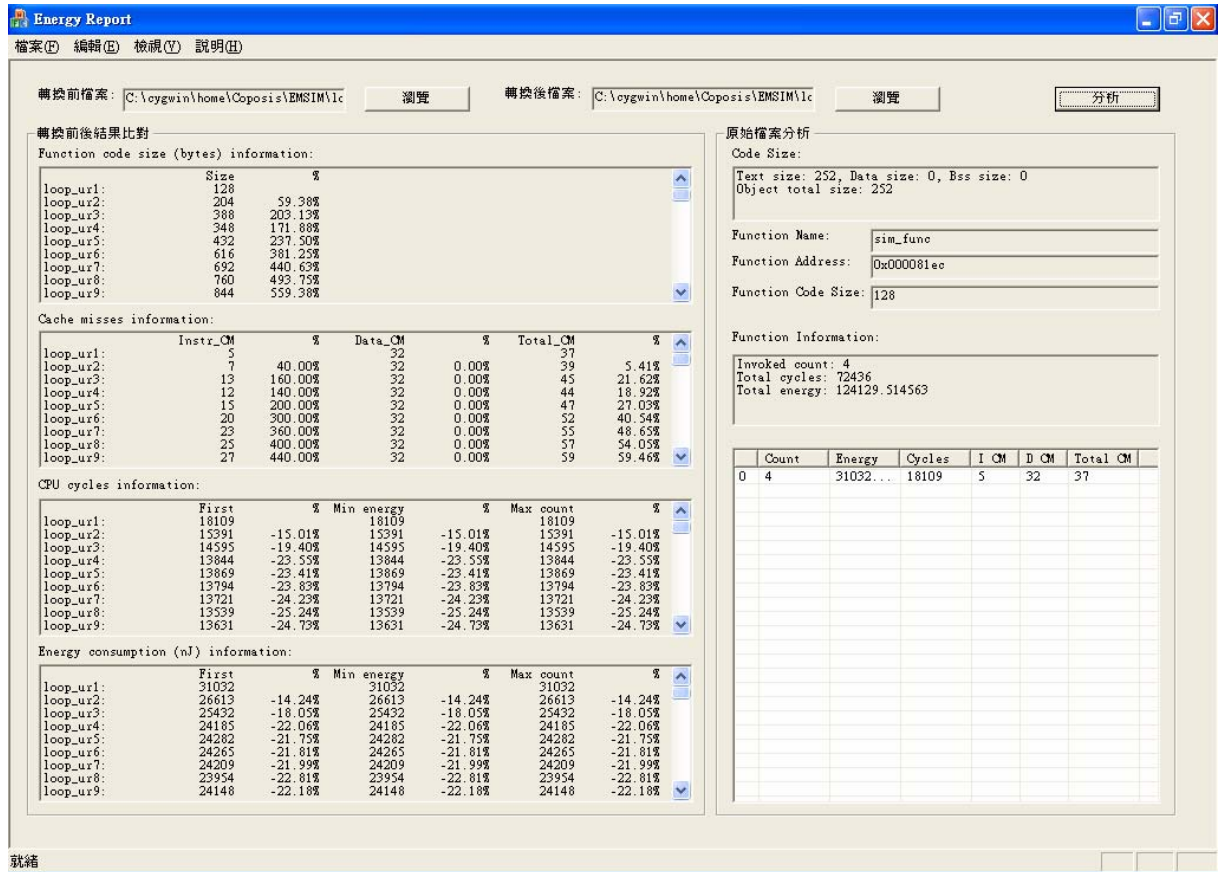


Figure 4-3 The execution result of the Energy Report program

Transformations in source code level lead to very different results depending on a number of factors, including the specific structure of code, the target architecture, and the parameters of the transformations, etc. [16]. But what compiler you use and of which the options you choose also have a significant impact. Because the EMSIM energy simulator is adopted, the target architecture is fixed in our experimental framework. Table 4-2 shows the target architecture in our experimental framework. In addition, we adopt gcc 2.95.3 as our cross compiler and use optimization level -O0 and option -fomit-frame-pointer to compile the

source code files of our experiments. Hence, the experiments of the transformations are designed to observe the results between the specific structure of code and the different parameters of the transformations in this thesis.

Table 4-2 The target architecture of our experimental framework

The component of the target architecture	Description
The processor	StrongARM SA-1100 (CPU clock: 206 MHz)
The cache architecture	32-way set-associative instruction cache: Cache size: 16 KB Cache line size: 32 bytes Replacement policy: round robin 32-way set-associative write-back data cache: Cache size: 16 KB Cache line size: 32 bytes Replacement policy: round robin

4.2 Data Transformations

In this section, the experiment of only one data transformation which is common sub-expression elimination is designed to verify its energy-efficiency.

4.2.1 Common Sub-expression Elimination

Original C source code	Transformed C source code
<pre>void sim_func(void) { int a, b, c, d, e, f; a=e/f; b=21+a; c=14+d; d=e/f; }</pre>	<pre>void sim_func(void) { int a, b, c, d, e, f; int _testTmp0; _testTmp0=e/f; a=_testTmp0; b=21+a; c=14+d; d=_testTmp0; }</pre>

Figure 4-4 C source code of common sub-expression elimination for Exp#1.1

We use the JuanCSE pass [36] which is free released to do common sub-expression elimination to transform the original C source code. The C source code before and after transformation are shown in Figure 4-4.

4.3 Loop Transformations

In this section, a number of experiments of loop transformations are designed to verify their energy-efficiency, including loop fusion, loop fission, loop reversal, loop inversion, loop interchange, loop unrolling and loop unswitching.

4.3.1 Loop Fusion

Original C source code	Transformed C source code
<pre>void sim_func(void) { int i, a[410], b[410]; for (i=0;i<410;i++) a[i]=41; for (i=0;i<410;i++) b[i]=14; }</pre>	<pre>void sim_func(void) { int i, a[410], b[410]; for (i=0;i<410;i++) { a[i]=41; b[i]=14; } }</pre>

Figure 4-5 C source code of loop fusion for Exp#2.1

As shown in Figure 4-5, we design an experiment in which the simulation function has two loops. After adding the code size of the two loop bodies, we find that the size is smaller than the size of the instruction cache, so we can use loop fusion transformation to reduce the loop overhead but not increase the number of the instruction cache misses. Hence, it is expected that the energy consumption is reduced after transformation.

4.3.2 Loop Fission

In our experimental framework, the size of the instruction cache is 16 KB. As shown in Figure 4-6, we design a simple experiment in which the loop body size in the simulation

function of the original C source code is larger than 16 KB, so there are not only compulsory misses but also capacity misses happened when the loop is executed. We then apply loop fission transformation to the loop of the original C source code to break down the loop into two ones of which the body size is a half of the original one. It is expected that after transformation, the number of the instruction cache misses are reduced remarkably resulting in reduced energy consumption.

Original C source code	Transformed C source code
<pre>void sim_func(void) { int i, a[410], b[410], c[410], d[410]; for (i=0;i<410;i++) { //the loop body size: 21008 } }</pre>	<pre>void sim_func(void) { int i, a[410], b[410], c[410], d[410]; for (i=0;i<410;i++) { // the loop body size: 10504 } for (i=0;i<410;i++) { // the loop body size: 10504 } }</pre>

Figure 4-6 C source code of loop fission for Exp#2.2

4.3.3 Loop Reversal

In section 3.3.3, it is expected that energy consumption will be reduced because loop reversal causes the original ADD/CMP instruction pair to be replaced by a single SUBS instruction on the ARM. We design a simple experiment to verify its hypothesis, as shown in Figure 4-7.

Original C source code	Transformed C source code
<pre>void sim_func(void) { int i, a[400], b[400]; for (i=0;i<400;i++) a[i]=b[i]+14; }</pre>	<pre>void sim_func(void) { int i, a[400], b[400]; for (i=399;i>=0;i--) a[i]=b[i]+14; }</pre>

Figure 4-7 C source code of loop reversal for Exp#2.3

4.3.4 Loop Inversion

As shown in Figure 4-8, Exp#2.4 is designed to verify energy-efficiency of loop inversion. By transforming a *for* loop to a *do-while* loop, it is expected to reduce the number of the instructions executed, leading to reduced energy consumption.

Original C source code	Transformed C source code
<pre>void sim_func(void) { int i, a[210]; for (i=0;i<210;i++) a[i]=41; }</pre>	<pre>void sim_func(void) { int i, a[210]; i=0; do { a[i]=41; i++; } while (i<210); }</pre>

Figure 4-8 C source code of loop inversion for Exp#2.4

4.3.5 Loop Interchange

Original C source code	Transformed C source code
<pre>void sim_func(void) { int i, j; char a[16][512]; for (j=0;j<=511;j++) for (i=0;i<=15;i++) a[i][j]=41; }</pre>	<pre>void sim_func(void) { int i, j; char a[16][512]; for (i=0;i<=15;i++) for (j=0;j<=511;j++) a[i][j]=41; }</pre>

Figure 4-9 C source code of loop interchange for Exp#2.5.a

This transformation is very useful to reduce conflict misses. In our experiments, we focus on the case in which the array size is smaller than the size of the data cache in our target architecture. Figure 4-9 shows the first experiment of this transformation. When executing the loop of the original program of Exp#2.5.a, it will result in compulsory misses in the same set of the data cache because of the row size of the two dimensional array is 512 Bytes. Conflict

misses will not be happened, because the number of the columns in this array is equal to 16 which is smaller than the number of the ways. Hence, it is expected that the energy consumption will not be changed after transformation.

In the second experiment, we use the same row size of the array as the first experiment, as shown in Figure 4-10. But the number of the columns is greater than the number of the ways; as a result, it will result in conflict misses when the 33rd column of the array is accessed in the original loop. Hence, it is useful to do loop interchange to eliminate the conflict misses resulting in reduced energy consumption.

Original C source code	Transformed C source code
<pre>void sim_func(void) { int i, j; char a[33][512]; for (j=0;j<=511;j++) for (i=0;i<=32;i++) a[i][j]=41; }</pre>	<pre>void sim_func(void) { int i, j; char a[33][512]; for (i=0;i<=32;i++) for (j=0;j<=511;j++) a[i][j]=41; }</pre>

Figure 4-10 C source code of loop interchange for Exp#2.5.b

4.3.6 Loop Unrolling

This transformation has a parameter, the unrolling factor. Because the unrolling factor has a significant impact on energy consumption, we design a C source code file generator program to generate 100 files of which the unrolling factor are 1 to 100 respectively.

As shown in Figure 4-11, the left is the original C source code of which the unrolling factor is 1 and the right is the transformed C source code of which the unrolling factor is U (2~100). It is noted that when U is not a divisor of the loop counts, it is necessary to copy the original loop and put the duplicated loop below the original one to complete the operations; otherwise, the loop copy can be eliminated.

Original C source code	Transformed C source code
<pre>void sim_func(void) { int i; char a[500], b[500]; for (i=0;i<500;i++) { a[i]=14; b[i]=a[i]*4+21; } }</pre>	<pre>void sim_func(void) { int i; char a[500], b[500]; for (i=0;i<501-U;i++) { a[i]=14; b[i]=a[i]*4+21; } for (;i<500;i++) { a[i]=14; b[i]=a[i]*4+21; } }</pre>

Figure 4-11 C source code of loop unrolling for Exp#2.6

4.3.7 Loop Unswitching

Original C source code	Transformed C source code
<pre>void sim_func(void) { int i, j, a[200]; j=21; for (i=0;i<200;i++) { if (j%4==0) a[i]=140; } }</pre>	<pre>void sim_func(void) { int i, j, a[200]; j=21; if (j%4==0) { for (i=0;i<200;i++) a[i]=140; } }</pre>

Figure 4-12 C source code of loop unswitching for Exp#2.7.a

Original C source code	Transformed C source code
<pre>void sim_func(void) { int i, j, a[200]; j=21; for (i=0;i<200;i++) { if (j%4==0) a[i]=140; else a[i]=210; } }</pre>	<pre>void sim_func(void) { int i, j, a[200]; j=21; if (j%4==0) { for (i=0;i<200;i++) a[i]=140; } else { for (i=0;i<200;i++) a[i]=210; } }</pre>

Figure 4-13 C source code of loop unswitching for Exp#2.7.b

It is expected that this transformation reduces energy consumption, but it may increase code size due to the loop copy for *else* parts of conditional. We design two experiments for this transformation to evaluate the impact on the code size. One has only *if* part and another has *else* part, as shown in Figure 4-12 and Figure 4-13, respectively.

4.4 Control Structures and Operators Transformations

In this section, the conditional sub-expression reordering of the control structures and operators transformations is discussed.

4.4.1 Conditional Sub-expression Reordering

This transformation is very simple to understand its operational principle. It reorders the conditional sub-expressions by their probability to reduce the number of the instructions executed. Hence, it is expected that the energy consumption will be reduced.

Original C source code	Transformed C source code
<pre>void sim_func(void) { int i, a; a=0; for (i=0;i<210;i++) if (i%3!=0 && i%2==0 i<105) a++; }</pre>	<pre>void sim_func(void) { int i, a; a=0 for (i=0;i<210;i++) if (i<105 i%2==0 && i%3!=0) a++; }</pre>

Figure 4-14 C source code of conditional sub-expression reordering for Exp#3.1

4.5 Procedural Transformations

A number of experiments of the procedural transformations which include procedure inlining, procedural integration and loop embedding are designed to evaluate their energy-efficiency in this section.

4.5.1 Procedure Inlining

Original C source code	Transformed C source code
<pre> int x, y; int sim_funcA(void) { int result; result=x*2+21; result=result+y*+41; result=result*result+2; result=result/2; return result; } void sim_func(void) { int i, a[100]; x=210; y=140; a[0]=sim_funcA(); for (i=1;i<98;i++) { x=x+i*41; y=y*14; a[i]=sim_funcA(); } x=21; y=14; a[98]=sim_funcA(); x=14; y=21; a[99]=sim_funcA(); } </pre>	<pre> int x, y; void sim_func(void) { int i, a[100]; int suif_tmp0, suif_tmp1, suif_tmp2; x=210; y=140; { int result; suif_tmp0=result; } a[0]=suif_tmp0; for (i=1;i<98;i++) { { int suif_tmp00; x=x+i*41; y=y*14; { int result; suif_tmp00=result; } a[i]=suif_tmp00; } } x=21; y=14; { int result; suif_tmp1=result; } a[98]=suif_tmp1; x=14; y=21; { int result; suif_tmp2=result; } a[99]=suif_tmp2; } </pre>

Figure 4-15 C source code of procedure inlining for Exp#4.1

As shown in Figure 4-15, the simulation function in the original C source code has four call sites at which the function ‘sim_funcA’ is invoked. We use the JuanInlining pass [36] which is free released to do procedure inlining to transform the original C source code.

4.5.2 Procedure Integration

Original C source code	Transformed C source code
<pre> int x, y; int sim_funcA(void) { int result; result=x*2+21; result=result+y*+41; result=result*result+2; result=result/2; return result; } void sim_func(void) { int i, a[100]; x=210; y=140; a[0]=sim_funcA(); for (i=1;i<98;i++) { x=x+i*41; y=y*14; a[i]=sim_funcA(); } x=21; y=14; a[98]=sim_funcA(); x=14; y=21; a[99]=sim_funcA(); } </pre>	<pre> int x, y; int sim_funcA(void) { int result; result=x*2+21; result=result+y*+41; result=result*result+2; result=result/2; return result; } void sim_func(void) { int i, a[100]; x=210; y=140; a[0]=sim_funcA(); for (i=1;i<98;i++) { { int suif_tmp00; x=x+i*41; y=y*14; { int result; suif_tmp00=result; } a[i]=suif_tmp00; } } x=21; y=14; a[98]=sim_funcA(); x=14; y=21; a[99]=sim_funcA(); } </pre>

Figure 4-16 C source code of procedure integration for Exp#4.2

Procedure integration is a general version of procedure inlining. It can decide which call site to do integration. As shown in Figure 4-16, we design Exp#4.2 which uses the same original file as Exp#4.1. But we only select the call site which is in the loop to do integration. Hence, it is expected that this transformation not only reduce energy consumption effectively but also control the increased code size within a reasonable range.

4.5.3 Loop Embedding

This transformation is expected to reduce energy consumption. We design a simple experiment to verify its energy-efficiency.

Original C source code	Transformed C source code
<pre>int sim_funcA(int x, int y, int c) { return x*x+y*y+c; } void sim_func(void) { int i, x, y, a[100]; x=140; y=210; for (i=0;i<100;i++) a[i]=sim_funcA(x, y, i); }</pre>	<pre>void sim_funcAA(int x, int y, int *a) { int i; for (i=0;i<100;i++) a[i]=x*x+y*y+i; } void sim_func(void) { int i, x, y, a[100]; x=140; y=210; sim_funcAA(x, y, a); }</pre>

Figure 4-17 C source code of loop embedding for Exp#4.3

4.6 ISA-specific Transformations

In this section, a number of experiments of ISA-specific transformations are designed to verify their energy-efficiency on ARM ISA.

4.6.1 Arrays Declaration Sorting

We design two experiments to observe the results on ARM ISA. Figure 4-18 and Figure

4-19 show the original and transformed source code for the two experiments respectively.

Original C source code	Transformed C source code
<pre>void sim_func(void) { int i, a[305], b[210], c[110]; for (i=0;i<305;i++) a[i]=41; for (i=0;i<210;i++) b[i]=41; for (i=0;i<110;i++) c[i]=41; }</pre>	<pre>void sim_func(void) { int i, b[210], c[110], a[305]; for (i=0;i<305;i++) a[i]=41; for (i=0;i<210;i++) b[i]=41; for (i=0;i<110;i++) c[i]=41; }</pre>

Figure 4-18 C source code of arrays declaration sorting for Exp#5.1.a

Original C source code	Transformed C source code
<pre>void sim_func(void) { int i, a[440], b[220], c[140], d[70]; for (i=0;i<440;i++) a[i]=41; for (i=0;i<220;i++) b[i]=41; for (i=0;i<140;i++) c[i]=41; for (i=0;i<70;i++) d[i]=41; }</pre>	<pre>void sim_func(void) { int i, b[220], c[140], d[70], a[440]; for (i=0;i<440;i++) a[i]=41; for (i=0;i<220;i++) b[i]=41; for (i=0;i<140;i++) c[i]=41; for (i=0;i<70;i++) d[i]=41; }</pre>

Figure 4-19 C source code of arrays declaration sorting for Exp#5.1.b

4.6.2 Dummy Variables Insertion

As shown in Figure 4-20, this transformation insert dummy variables to reduce the number of the instructions executed to calculate the bases addresses of the arrays; as a result, it is expected to reduce energy consumption on ARM ISA.

Original C source code	Transformed C source code
<pre>void sim_func(void) { int i, a[440], b[220], c[140], d[70]; for (i=0;i<440;i++) a[i]=41; for (i=0;i<220;i++) b[i]=41; for (i=0;i<140;i++) c[i]=41; for (i=0;i<70;i++) d[i]=41; }</pre>	<pre>void sim_func(void) { int i, b[220], c[140]; volatile int dummy1[2]; int d[70], a[440]; for (i=0;i<440;i++) a[i]=41; for (i=0;i<220;i++) b[i]=41; for (i=0;i<140;i++) c[i]=41; for (i=0;i<70;i++) d[i]=41; }</pre>

Figure 4-20 C source code of dummy variables insertion for Exp#5.2

4.6.3 Arrays Declaration Permutation

As shown in Figure 4-21, this transformation suitably permutes the order of the arrays declaration to reduce the number of the instructions executed to calculate the bases addresses of the arrays; as a result, it is expected to reduce energy consumption on ARM ISA.

Original C source code	Transformed C source code
<pre>void sim_func(void) { int i, a[440], b[220], c[140], d[70]; for (i=0;i<440;i++) a[i]=41; for (i=0;i<220;i++) b[i]=41; for (i=0;i<140;i++) c[i]=41; for (i=0;i<70;i++) d[i]=41; }</pre>	<pre>void sim_func(void) { int i, d[70], c[140], b[220], a[440]; for (i=0;i<440;i++) a[i]=41; for (i=0;i<220;i++) b[i]=41; for (i=0;i<140;i++) c[i]=41; for (i=0;i<70;i++) d[i]=41; }</pre>

Figure 4-21 C source code of arrays declaration permutation for Exp#5.3

Chapter 5

Results and Analyses

In this chapter, we list the results of the experiments which are designed in Chapter 4. From the results, we try to analyze the relationship between energy consumption and side effects such as code size and performance.

Table 5-1 The definition of notations

Notation	Definition
$\lfloor x \rfloor$	The greatest integer less than or equal to x
$\lceil x \rceil$	The least integer greater than or equal to x
U	Unrolling factor
LO	Loop overhead
LB	Loop body
ICM	Instruction cache miss
DCM	Data cache miss
N_{LC}	The loop counts
N_{DCM}	The number of data cache misses
N_{DCM}'	The number of data cache misses after transformation
S_{ICL}	The size of the instruction cache line
S_{LO}	The loop overhead size
S_{LB}	The loop body size
E_{LO}	The energy consumption of the instructions executed for the loop overhead
E_{LB}	The energy consumption of the instructions executed for the loop body
E_{cmp}	The energy consumption of the compare operation before transformation
E_{cmp}'	The energy consumption of the compare operation after transformation
E_{ICM}	The energy consumption of memory access for every instruction cache miss
E_{DCM}	The energy consumption of memory access for every data cache miss
E_{ori}	The energy consumption of the affected code before transformation
E_{aft}	The energy consumption of the affected code after transformation
ΔE	The energy consumption savings after transformation ($\Delta E = E_{aft} - E_{ori}$)

In order to simplify our analyses, we take assumptions for some transformations. After analyses, we will show the limitations of transformations resulting from compilers and ISA if necessary. We also create energy equations to express the energy consumption savings for transformations if possible. In our energy equations, we only consider the three main factors on energy consumption, including the energy consumption of the instructions executed and the energy consumption of memory access for the instruction and data cache misses. Table 5-1 lists the definition of notations used in the following sections.

5.1 Data Transformations

In this section, the experimental results of the data transformations are listed and analyzed.

5.1.1 Common Sub-expression Elimination

Table 5-2 The result of Exp#1.1

Parameters	Original	Transformed	%
Code Size (bytes)	80	76	-5.00
Instruction Cache Misses	4	3	-25.00
Data Cache Misses	1	1	0.00
CPU Cycles	308	240	-22.08
Energy Consumption (nJ)	747	613	-17.92

Table 5-3 The definition of notations used in Section 5.1.1

Notation	Definition
N_{CSE}	The number of common sub-expression
E_{CSE}	The energy consumption of the instructions executed for the common sub-expression
E_{LDR_CSE}	The energy consumption of loading the computation result of the common sub-expression
E_{STR_CSE}	The energy consumption of storing the computation result of the common sub-expression

As shown in Table 5-2, the number of the data cache misses is not affected. But it is noted that it may result in the increased number of the data cache misses due to the introduced variables used in storing the computation result of common sub-expressions. In addition, the code size and the number of the instruction cache misses are reduced due to the reduced number of the instructions for recomputing; as a result, the CPU cycles consumed and the energy consumption are reduced. Table 5-3 lists the definition of notations used in this section. The energy consumption savings can be expressed as:

$$\Delta E \approx (1 - N_{CSE}) \times E_{CSE} + N_{CSE} \times E_{LDR_CSE} + E_{STR_CSE} \quad \text{Eq. (5.1)}$$

5.2 Loop Transformations

In this section, the experimental results of the loop transformations are listed and analyzed.



5.2.1 Loop Fusion

Assumption: the size of the loop which is the fusion of several loops is equal to or less than the size of the instruction cache.

Table 5-4 The result of Exp#2.1

Parameters	Original	Transformed	%
Code Size (bytes)	176	124	-29.55
Instruction Cache Misses	7	5	-28.57
Data Cache Misses	103	103	0.00
CPU Cycles	20241	14047	-30.60
Energy Consumption (nJ)	36288	25845	-28.78

As shown in Table 5-4, the number of the data cache misses is not affected. The code size and the number of the instruction cache misses are reduced due to the reduced number of

the instructions of the loop overheads; as a result, the CPU cycles consumed and the energy consumption are reduced. Table 5-5 lists the definition of notations used in this section. The energy consumption of the loops before and after transformation can be expressed as follows:

$$E_{ori} = N_l \times N_{LC} \times E_{LO} + N_{LC} \times \sum_{i=0}^{N_l-1} E_{LBi} + \left[\frac{N_l \times S_{LO} + \sum_{i=0}^{N_l-1} S_{LBi}}{S_{ICL}} \right] \times E_{ICM} + N_{DCM} \times E_{DCM} \quad \text{Eq. (5.2)}$$

$$E_{aft} = N_{LC} \times E_{LO} + N_{LC} \times \sum_{i=0}^{N_l-1} E_{LBi} + \left[\frac{S_{LO} + \sum_{i=0}^{N_l-1} S_{LBi}}{S_{ICL}} \right] \times E_{ICM} + N_{DCM} \times E_{DCM} \quad \text{Eq. (5.3)}$$

Because N_{DCM}' is equal to N_{DCM} , the energy consumption savings can be derived as:

$$\Delta E \approx (1 - N_l) \times N_{LC} \times E_{LO} + \left[\frac{(1 - N_l) \times S_{LO}}{S_{ICL}} \right] \times E_{ICM} \quad \text{Eq. (5.4)}$$

Table 5-5 The definition of notations used in Section 5.2.1

Notation	Definition
LB_i	The loop body of the i th loop
N_l	The number of the loops which have the same loop counts
S_{LBi}	The loop body size of the i th loop
E_{LBi}	The energy consumption of the instructions executed for the loop body of the i th loop

5.2.2 Loop Fission

Assumption: the size of the loop before transformation is greater than the size of the instruction cache, and the size of the every loop after transformation is equal to or less than the size of the instruction cache.

As shown in Table 5-6, the number of the data cache misses is not affected. The code size is increased due to the increased number of the instructions of the loop overheads. It is very noted that the number of the instruction cache misses are reduced remarkably due to the

reduced number of the capacity misses; as a result, the CPU cycles consumed and the energy consumption are reduced.

Table 5-6 The result of Exp#2.2

Parameters	Original	Transformed	%
Code Size (bytes)	21100	21168	0.32
Instruction Cache Misses	270193	663	-99.75
Data Cache Misses	206	206	0.00
CPU Cycles	8917588	4606762	-48.34
Energy Consumption (nJ)	20210333	7526457	-62.76

5.2.3 Loop Reversal

As shown in Table 5-7, the code size is almost unchanged after loop reversal transformation. The number of the instruction and the data cache misses are not affected. From the assembly code files, we find that the CPU cycles consumed are reduced because comparing with zero only needs one instruction and comparing to another value may need more than one instruction; as a result, the energy consumption may be reduced or not. However, it is noted that this transformation may reduce the dependence of the codes in loop body to make other transformations applied. The energy consumption savings can be express

as: $\Delta E \approx E_{cmp}' - E_{cmp}$ Eq. (5.5)

Table 5-7 The result of Exp#2.3

Parameters	Original	Transformed	%
Code Size (bytes)	116	112	-3.45
Instruction Cache Misses	5	5	0.00
Data Cache Misses	101	101	0.00
CPU Cycles	13315	12514	-6.02
Energy Consumption (nJ)	24667	23379	-5.22

Limitation: it is affected by the compiler and the ISA used if you want to reduce energy consumption.

5.2.4 Loop Inversion

As shown in Table 5-8, the code size and the number of the instruction cache misses are almost unchanged after loop inversion transformation. The number of the data cache misses is not affected. The CPU cycles consumed are reduced due to the reduced compare and branch instructions; as a result, the energy consumption is reduced.

Table 5-8 The result of Exp#2.4

Parameters	Original	Transformed	%
Code Size (bytes)	80	76	-5.00
Instruction Cache Misses	4	3	-25.00
Data Cache Misses	27	27	0.00
CPU Cycles	4711	4685	-0.55
Energy Consumption (nJ)	8599	8446	-1.77

5.2.5 Loop Interchange

As shown in Table 5-9 and Table 5-10, the code size and the number of the instruction cache misses are not affected after loop interchange transformation. In Exp#2.5.a, as the same as expected, the number of the data cache misses is not affected due to no changes in the conflict misses after transformations. But the CPU cycles consumed and the energy consumption are not the same as expected. From the assembly code, we find that comparing the loop bound with 15 needs only one instruction but comparing the loop bound with 511 needs three instructions. Because comparing with 511 is in the outer loop before transformation but in the inner loop after transformation; as a result, it will increase the number of the instructions executed to do compare operations. Hence, it increases the CPU

cycles consumed and the energy consumption slightly.

In Exp#2.5.b, as the same as expected, the number of the data cache misses is reduced due to the conflict misses eliminated; as a result, the CPU cycles consumed and the energy consumption are reduced remarkably.

We conclude that when the size of the array is equal to or less than the size of the data cache, loop interchange can reduce energy consumption if the conflict misses exist. On the other hand, if the conflict misses do not exist, it may reduce or increase energy consumption according to the number of the instructions executed to do compare operations reduced or increased. When the size of the array is greater than the size of the data cache, this transformation is very useful to reduce energy consumption due to the capacity and conflict misses eliminated.

Table 5-9 The result of Exp#2.5.a

Parameters	Original	Transformed	%
Code Size (bytes)	204	204	0.00
Instruction Cache Misses	7	7	0.00
Data Cache Misses	257	257	0.00
CPU Cycles	289429	291427	0.69
Energy Consumption (nJ)	480971	483739	0.58

Table 5-10 The result of Exp#2.5.b

Parameters	Original	Transformed	%
Code Size (bytes)	204	204	0.00
Instruction Cache Misses	7	7	0.00
Data Cache Misses	16905	530	-96.86
CPU Cycles	974301	601200	-38.29
Energy Consumption (nJ)	2168591	998158	-53.97

5.2.6 Loop Unrolling

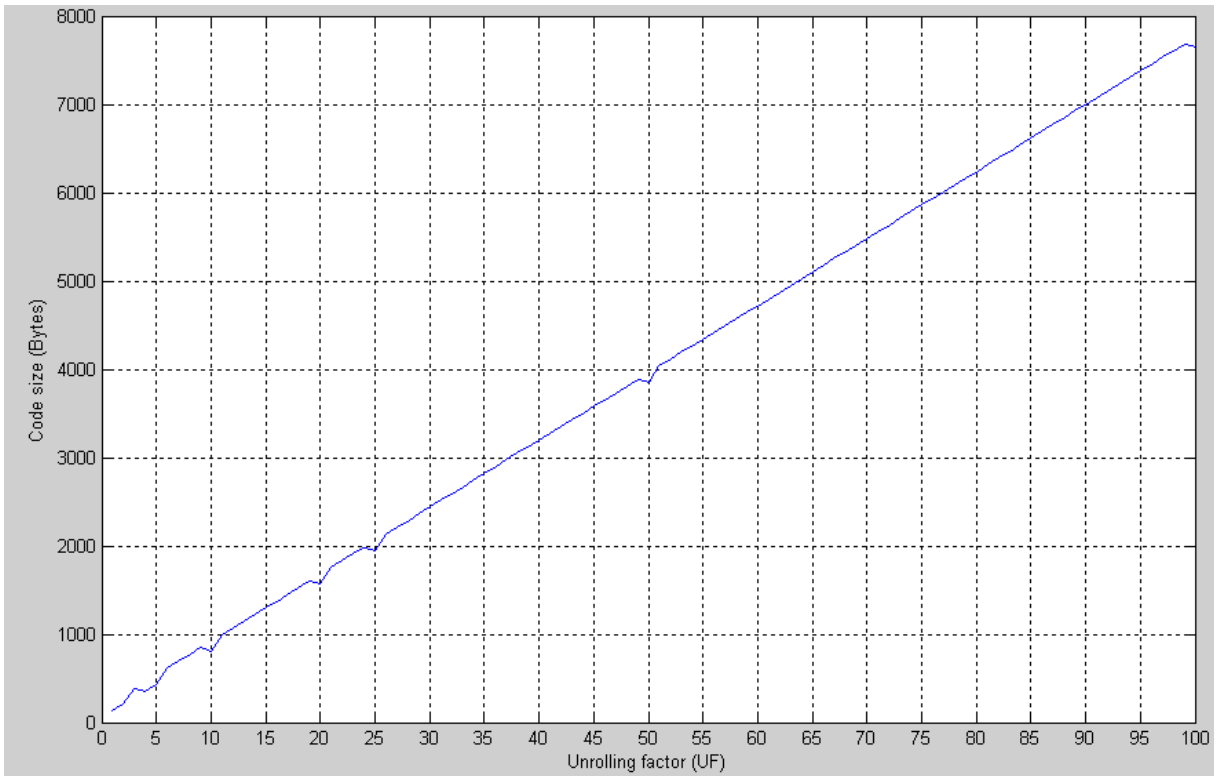


Figure 5-1 The results of the code size in Exp#2.6

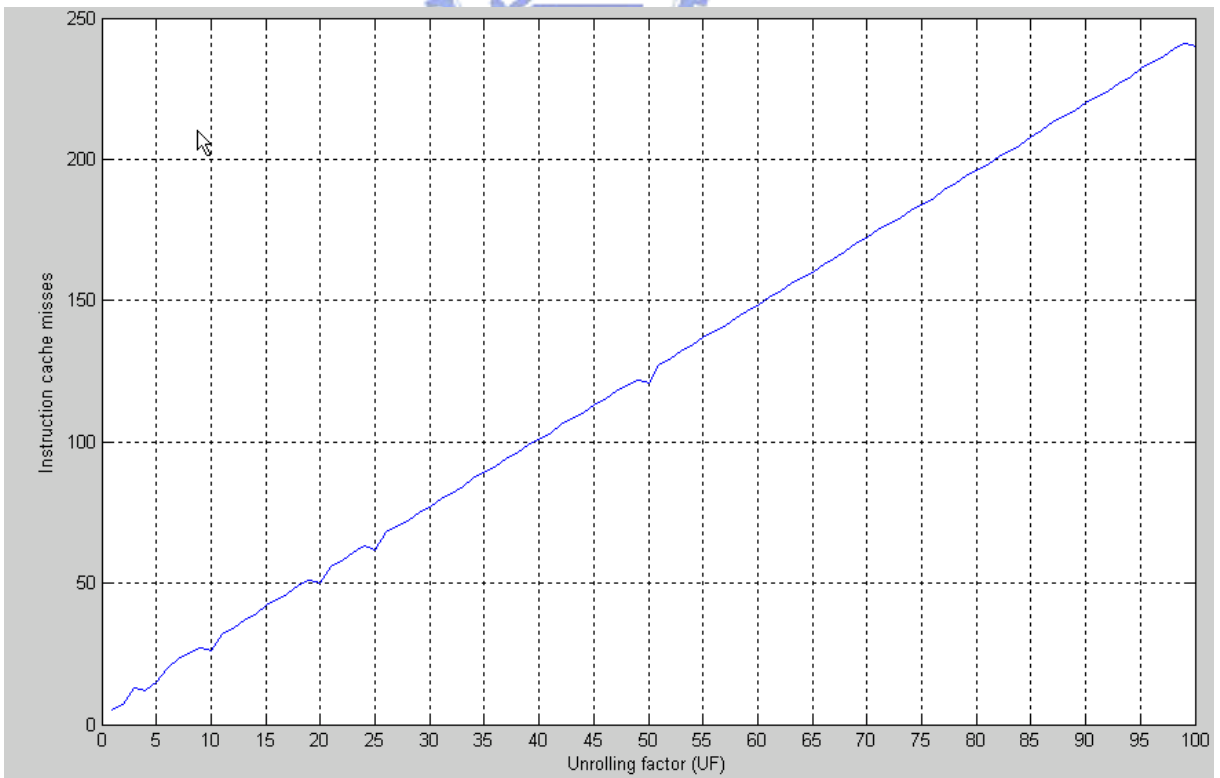


Figure 5-2 The results of the instruction cache misses in Exp#2.6

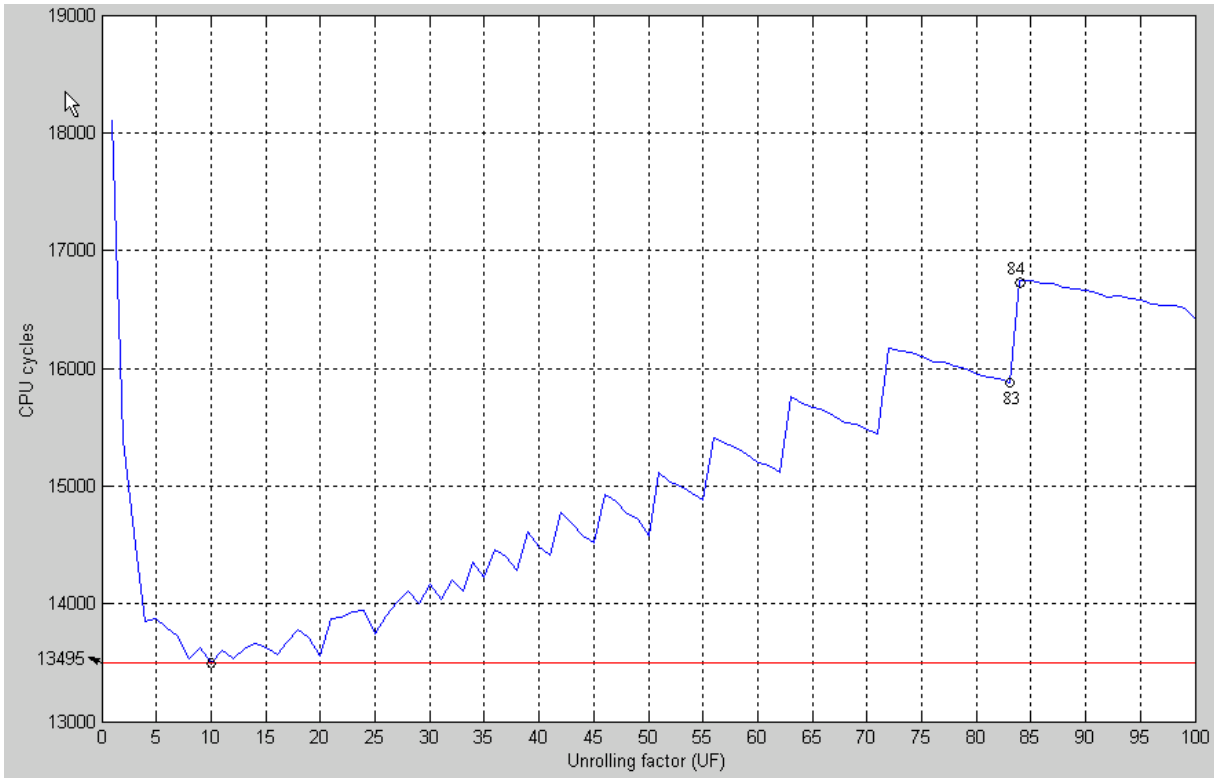


Figure 5-3 The results of the CPU cycles in Exp#2.6

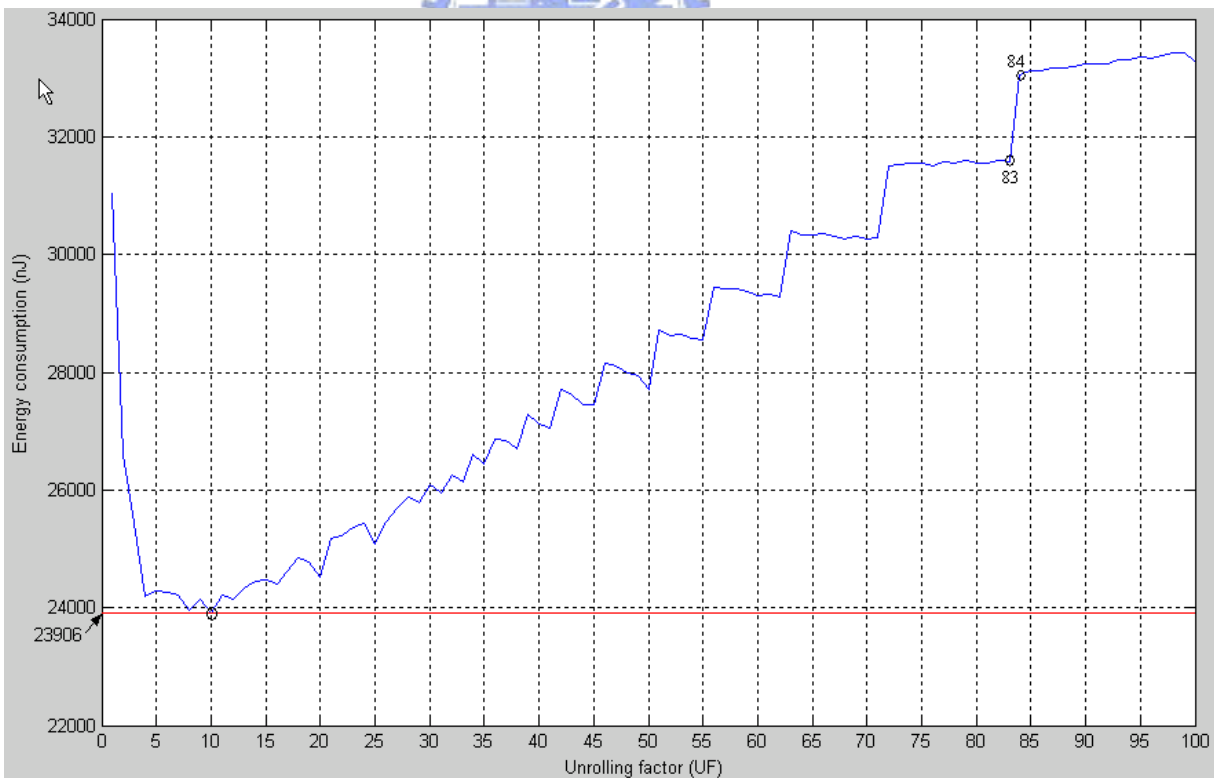


Figure 5-4 The results of the energy consumption in Exp#2.6

Assumption: The loop size after unrolling is equal to or less than the size of the instruction cache.

The unrolling factor of loop unrolling transformation has a significant impact on energy consumption. Hence, energy equations are created for us to get the relationship between unrolling factor and energy consumption. Because of the assumption, we can know that there are only compulsory misses needed to be considered in our equations. The energy consumption of the loop before and after transformation can be expressed as follows:

$$E_{ori} = N_{LC} \times (E_{LO} + E_{LB}) + \left\lceil \frac{S_{LO} + S_{LB}}{S_{ICL}} \right\rceil \times E_{ICM} + N_{DCM} \times E_{DCM} \quad \text{Eq. (5.6)}$$

$(N_{LC} \% U) \neq 0$:

$$E_{aft} = \left\lceil \frac{N_{LC}}{U} \right\rceil + (N_{LC} \% U) \times E_{LO} + N_{LC} \times E_{LB} + \left\lceil \frac{(S_{LO} + S_{LB} \times U) + (S_{LO} + S_{LB})}{S_{ICL}} \right\rceil \times E_{ICM} + N_{DCM} \times E_{DCM} \quad \text{Eq. (5.7)}$$

$(N_{LC} \% U) = 0$:

$$E_{aft} = \left\lceil \frac{N_{LC}}{U} \right\rceil \times E_{LO} + N_{LC} \times E_{LB} + \left\lceil \frac{S_{LO} + S_{LB} \times U}{S_{ICL}} \right\rceil \times E_{ICM} + N_{DCM} \times E_{DCM} \quad \text{Eq. (5.8)}$$

Because N_{DCM}' is equal to N_{DCM} , the energy consumption savings can be derived as follows:

$$(N_{LC} \% U) \neq 0: \Delta E \approx \left\lceil \frac{N_{LC}}{U} \right\rceil + (N_{LC} \% U) - N_{LC} \times E_{LO} + \left\lceil \frac{S_{LO} + S_{LB} \times U}{S_{ICL}} \right\rceil \times E_{ICM} \quad \text{Eq. (5.9)}$$

$$(N_{LC} \% U) = 0: \Delta E \approx \left\lceil \frac{N_{LC}}{U} \right\rceil - N_{LC} \times E_{LO} + \left\lceil \frac{S_{LB} \times (U - 1)}{S_{ICL}} \right\rceil \times E_{ICM} \quad \text{Eq. (5.10)}$$

From the above equations, we can find that if the unrolling factor is not a divisor of the loop counts, the $(N_{LC} \% U)$ may result in a very critical issue when the unrolling factor is big. We also suppose that the minimum energy consumption will happen at the unrolling factor which is a divisor of the loop counts, because the eliminated loop copy and its overhead.

As shown in Figure 5-1 and Figure 5-2, the number of the instruction cache misses is in proportion to the code size. When unrolling factor is a divisor of the loop counts, the code size

and the number of instruction cache misses drop abruptly due to the loop copy eliminated. As show in Figure 5-3 and Figure 5-4, the CPU cycles consumed and the energy consumption are reduced due to the reduced number of the instructions executed for the loop overhead. When the unrolling factor is equal to 10, the minimum energy consumption is achieved. Besides, we also find some sharp increase at some unrolling factor. This is due to the impact of $(N_{LC}\%U)$. For example, we find a sharp increase at UF=84. Because the reduced loop overhead is equal to $\lfloor 500/83 \rfloor + (500\%83) - 500 = -492$ at UF=83 and is equal to $\lfloor 500/84 \rfloor + (500\%84) - 500 = -415$ at UF=84, the energy consumption has a big increase from UF=83 to UF=84.

5.2.7 Loop Unswitching

Table 5-11. The result of Exp#2.7.a

Parameters	Original	Transformed	%
Code Size (bytes)	100	100	0.00
Instruction Cache Misses	4	3	-25.00
Data Cache Misses	1	1	0.00
CPU Cycles	4096	78	-98.10
Energy Consumption (nJ)	7006	214	-96.95

Table 5-12 The result of Exp#2.7.b

Parameters	Original	Transformed	%
Code Size (bytes)	128	168	31.25
Instruction Cache Misses	5	5	0.00
Data Cache Misses	26	26	0.00
CPU Cycles	5912	4519	-23.56
Energy Consumption (nJ)	10606	8276	-21.98

As shown in Table 5-11 and Table 5-12, the number of the data cache misses is not affected after loop unswitching transformation. The number of the instruction cache misses is almost unchanged. As the same as expected, the impacts on the code size are inconsistent in

Exp#2.7.a and Exp#2.7.b. It results from the loop copy for the *else* part of conditional. Because the number of the instructions executed is reduced, the CPU cycles consumed and the energy consumption are reduced.

5.3 Control Structures and Operators Transformations

In this section, the experimental results of the control structures and operators transformations are listed and analyzed.

5.3.1 Conditional Sub-expression Reordering

Assumption: every sub-expression of the conditional consumes the same energy.

As shown in Table 5-13, the code size and the number of the instruction cache misses are almost unchanged after conditional sub-expression reordering transformation. The number of the data cache misses is not affected. The CPU cycles consumed are reduced due to the reduced number of instructions executed for the sub-expressions computation; as a result, the energy consumption is reduced.

Table 5-13 The result of Exp#3.1

Parameters	Original	Transformed	%
Code Size (bytes)	140	136	-2.86
Instruction Cache Misses	5	5	0.00
Data Cache Misses	1	1	0.00
CPU Cycles	27050	10785	-60.13
Energy Consumption (nJ)	44075	18004	-59.15

5.4 Procedural Transformations

In this section, the experimental results of the procedural transformations are listed and

analyzed. Table 5-14 lists the definition of notations used in this section.

Table 5-14 The definition of notations used in Section 5.4

Notation	Definition
PO	Procedural overhead
PB	Procedural body
CS _i	The ith call site which invokes the same procedure
N _{CS}	The number of the call sites which invoke the same procedure
N _{P_CSi}	The number of the procedure invoked at the ith call site
E _{PO}	The energy consumption of the instructions executed for the procedural overhead
E _{PB}	The energy consumption of the instructions executed for the procedural body

5.4.1 Procedure Inlining

Before transformation, the code sizes of the simulation procedure and the procedure which the simulation procedure invoked are 288 and 140 bytes respectively; the numbers of the instruction cache misses of those are 10 and 4 respectively; the numbers of the data cache misses of those are 15 and 2 respectively. After transformation, the procedure invoked is eliminated. As shown in Table 5-15, the code size and the number of the instruction cache misses are increased due to the procedure inlining. The number of the data cache misses is not affected. The CPU cycles consumed are reduced due to the reduced overhead of the procedure invoked. Although the number of instruction cache misses is increased resulting in the increased energy consumption of memory access, the call site in the loop results in more energy consumption savings; as a result, the energy consumption is reduced. The energy consumption of the affected code before and after transformation can be expressed as follows:

$$E_{ori} \approx (E_{PO} + E_{PB}) \times \sum_{i=0}^{N_{CS}-1} N_{P_CSi} + \left[\frac{S_{PO} + S_{PB}}{S_{ICL}} \right] \times E_{ICM} + N_{DCM} \times E_{DCM} \quad \text{Eq. (5.11)}$$

$$E_{aft} \approx E_{PB} \times \sum_{i=0}^{N_{CS}-1} N_{P_CSi} + \left[\frac{N_{CS} \times S_{PB}}{S_{ICL}} \right] \times E_{ICM} + N_{DCM} \times E_{DCM} \quad \text{Eq. (5.12)}$$

If we ignore the possible slight changes in the number of the data cache misses, N_{DCM}' is equal

to N_{DCM} ; as a result, the energy consumption savings can be derived as:

$$\Delta E \approx -E_{PO} \times \sum_{i=0}^{N_{CS}-1} N_{P_CSi} + \left[\frac{-S_{PO} + (N_{CS} - 1) \times S_{PB}}{S_{ICL}} \right] \times E_{ICM} \quad \text{Eq. (5.13)}$$

Table 5-15 The result of Exp#4.1

Parameters	Original	Transformed	%
Code Size (bytes)	428	732	71.03
Instruction Cache Misses	14	24	71.43
Data Cache Misses	17	17	0.00
CPU Cycles	11023	10180	-7.65
Energy Consumption (nJ)	19058	17987	-5.62

5.4.2 Procedure Integration

Table 5-16 The result of Exp#4.2

Parameters	Original	Transformed	%
Code Size (bytes)	428	540	26.17
Instruction Cache Misses	14	18	28.57
Data Cache Misses	17	18	5.88
CPU Cycles	11023	10133	-8.07
Energy Consumption (nJ)	19058	17804	-6.58

The code size of the procedure which the simulation procedure invoked is 140 bytes, and the code sizes of the simulation function before and after transformation are 288 and 400 bytes respectively. The numbers of the instruction cache misses of the simulation procedure before and after transformation are 10 and 14 respectively, and those of the procedure invoked before and after transformation are the same values, 4. The numbers of the data cache misses of the simulation procedure before and after transformation are 15 and 16 respectively, and those of the procedure invoked before and after transformation are the same values, 2. As shown in Table 5-16, the code size and the number of the instruction cache misses are increased due to the procedure integration. The number of the data cache misses is increased

slightly. The CPU cycles consumed are reduced due to the reduced overhead of the procedure invoked. In this experiment, we only expand the procedure at the call site which is in the loop. Compared with Exp#4.1, we can only not get better energy consumption savings, but also smaller increased code size. We suppose that the first ‘m’ call sites are needed to do procedure integration, and the others only invoke the original procedure. Hence, the energy consumption of the affected code before transformation is the same as Eq. (5.11), and that after transformation can be expressed as:

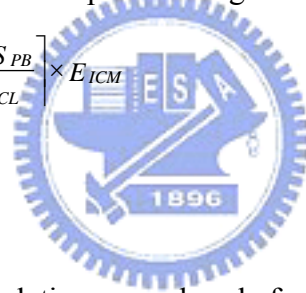
$$E_{aft} \approx E_{PO} \times \sum_{i=m}^{N_{CS}-1} N_{P_CSi} + E_{PB} \times \sum_{i=0}^{N_{CS}-1} N_{P_CSi} + \left[\frac{S_{PO} + (m+1) \times S_{PB}}{S_{ICL}} \right] \times E_{ICM} + N_{DCM}' \times E_{DCM}$$

Eq. (5.14)

If we ignore the possible slight changes in the number of the data cache misses, N_{DCM}' is equal to N_{DCM} ; as a result, the energy consumption savings can be derived as:

$$\Delta E \approx -E_{PO} \times \sum_{i=0}^{m-1} N_{P_CSi} + \left[\frac{m \times S_{PB}}{S_{ICL}} \right] \times E_{ICM}$$

Eq. (5.15)



5.4.3 Loop Embedding

The code sizes of the simulation procedure before and after transformation are 112 and 52 bytes respectively, and those of the procedure which the simulation procedure invoked before and after transformation are 68 and 124 bytes respectively. The numbers of the instruction cache misses of the simulation procedure before and after transformation are 5 and 3 respectively, and those of the procedure invoked before and after transformation are 2 and 4 respectively. The numbers of the data cache misses of the simulation procedure before and after transformation are 13 and 1 respectively, and those of the procedure invoked before and after transformation are 1 and 13 respectively. As shown in Table 5-17, the code size is almost unchanged after loop embedding transformation. The numbers of the instruction and the data cache misses are not affected, but in fact, the numbers of the instruction and the data cache misses may be affected slightly. The CPU cycles consumed are reduced due to the reduced

overhead of the procedure invoked; as a result, the energy consumption is reduced. If we ignore the possible slight changes in the numbers of the instruction and the data cache misses, the energy consumption savings can be derived as:

$$\Delta E \approx (1 - N_{LC}) \times E_{PO} \quad \text{Eq. (5.16)}$$

Table 5-17 The result of Exp#4.3

Parameters	Original	Transformed	%
Code Size (bytes)	180	176	-2.22
Instruction Cache Misses	7	7	0.00
Data Cache Misses	14	14	0.00
CPU Cycles	6857	4874	-28.92
Energy Consumption (nJ)	11893	8639	-27.36

5.5 ISA-specific Transformations

In this section, the experimental results of the ISA-specific transformations are listed and analyzed. Table 5-18 lists the definition of notations used in Section 5.5. Besides, we create general energy equations for Section 5.5.1-5.5.3 to use as follows:

$$E_{ori} = \sum_{i=0}^{N_a-1} (N_{BAi} \times E_{BAi}) + \left[\frac{\sum_{i=0}^{N_a-1} S_{BAi}}{S_{ICL}} \right] \times E_{ICM} + N_{DCM} \times E_{DCM} \quad \text{Eq. (5.17)}$$

$$E_{aft} = \sum_{i=0}^{N_a-1} (N_{BAi} \times E_{BAi}') + \left[\frac{\sum_{i=0}^{N_a-1} S_{BAi}'}{S_{ICL}} \right] \times E_{ICM} + N_{DCM}' \times E_{DCM} \quad \text{Eq. (5.18)}$$

$$\Delta E \approx \sum_{i=0}^{N_a-1} [N_{BAi} \times (E_{BAi}' - E_{BAi})] + \left[\frac{\sum_{i=0}^{N_a-1} (S_{BAi}' - S_{BAi})}{S_{ICL}} \right] \times E_{ICM} + (N_{DCM}' - N_{DCM}) \times E_{DCM} \quad \text{Eq. (5.19)}$$

Table 5-18 The definition of notations used in Section 5.5

Notation	Definition
a_i	The i th array in the simulation function
N_a	The number of local arrays in the simulation function
BA_i	The base address of a_i
N_{BAi}	The number of calculating the base address of a_i
S_{BAi}	The total code size of calculating the base address of a_i before transformation
S_{BAi}'	The total code size of calculating the base address of a_i after transformation
E_{BA}	The minimum energy consumption of calculating the base address (only take one instruction)
E_{BAi}	The energy consumption of calculating the base address of a_i before transformation
E_{BAi}'	The energy consumption of calculating the base address of a_i after transformation

5.5.1 Arrays Declaration Sorting

As shown in Table 5-19 and Table 5-20, the code size is increased after arrays declaration sorting transformation. The number of the instruction and the data cache misses are not affected. But in Exp#5.1.a and Exp#5.1.b, the impacts on the CPU cycles consumed and the energy consumption are inconsistent. Because N_{DCM}' is equal to N_{DCM} , according to Eq. (5.19), the energy consumption savings can be derived as:

$$\Delta E \approx \sum_{i=0}^{N_a-1} [N_{BAi} \times (E_{BAi}' - E_{BAi})] + \left[\frac{\sum_{i=0}^{N_a-1} (S_{BAi}' - S_{BAi})}{S_{ICL}} \right] \times E_{ICM} \quad \text{Eq. (5.20)}$$

Arrays declaration sorting puts the mostly accessed array on the top of the stack; as a result, it changes the offsets of the arrays in the stack, so it may increase the number of the instructions executed to calculate the base addresses of the arrays. Figure 5-5 shows the stack content of Exp#5.1.a. Before transformation, it only needs one instruction to calculate the base address of the every array. After transformation, it needs more than one instruction to calculate the base addresses of the array 'b' and 'c'. Hence, the CPU Cycles consumed are increased resulting in the increased energy consumption.

Table 5-19 The result of Exp#5.1.a

Parameters	Original	Transformed	%
Code Size (bytes)	224	232	3.57
Instruction Cache Misses	8	8	0.00
Data Cache Misses	79	79	0.00
CPU Cycles	13929	14249	2.30
Energy Consumption (nJ)	25328	25844	2.04

Table 5-20 The result of Exp#5.1.b

Parameters	Original	Transformed	%
Code Size (bytes)	304	308	1.32
Instruction Cache Misses	11	11	0.00
Data Cache Misses	109	109	0.00
CPU Cycles	20689	20609	-0.39
Energy Consumption (nJ)	37316	37191	-0.33

As shown in Figure 5-6, the transformation results in the two arrays 'b' and 'c' instead of the one array 'a' needed to take more than one instruction to calculate the base addresses in Exp#5.1.b. And from the assembly code, we find that the base address of the array 'a' is calculated 440 times and the base addresses of the arrays 'b' and 'c' are calculated 360 times (220 times +140 times) and we also find all of the base addresses of the arrays 'a' in original code, and 'b' and 'c' in transformed code needed to take two instructions to calculate; as a result, 80 instructions executed are saved. Hence, the CPU cycles consumed and the energy consumption are reduced.

From the above discussion, we conclude that this transformation does not always result in energy consumption due to ARM ISA, so we propose two transformations based on ARM ISA to improve the behavior of this transformation.

Limitation: it is affected by the ISA used and the strategies of the compiler to allocate stack

space for local variables.

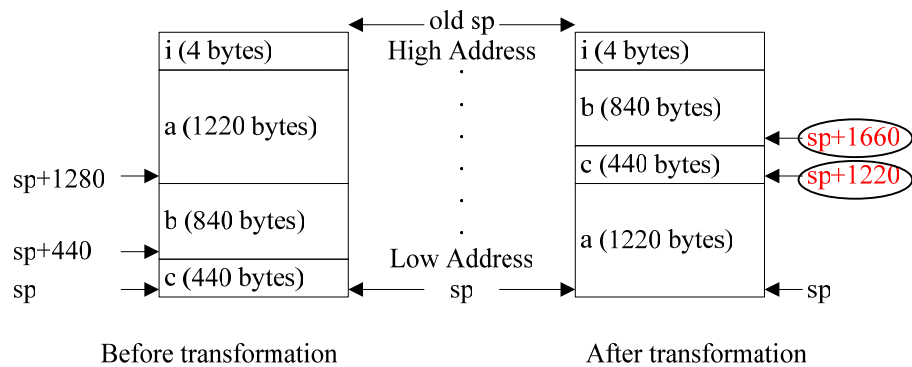


Figure 5-5 The stack content of Exp#5.1.a

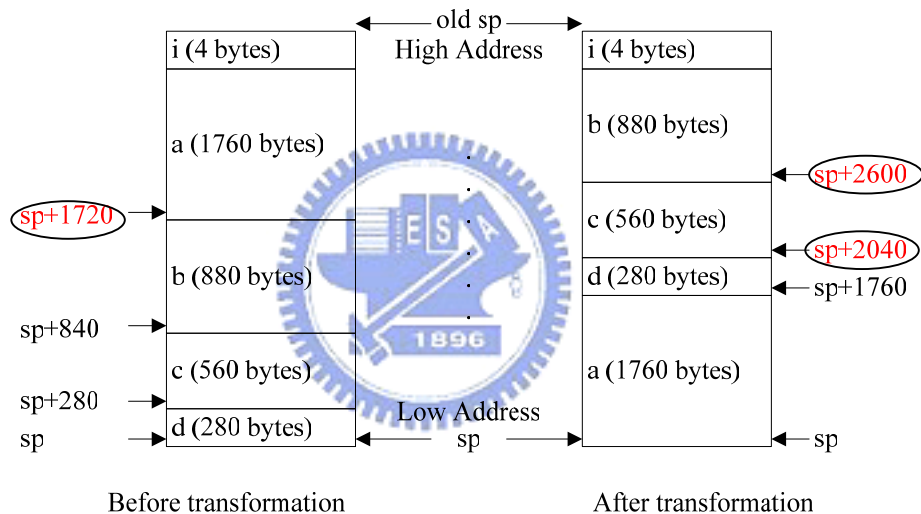


Figure 5-6 The stack content of Exp#5.1.b

5.5.2 Dummy Variables Insertion

Assumption: the modified data structures will not result in changes in the number of the conflict misses.

As shown in Table 5-21, the code size and the number of the instruction cache misses are reduced after dummy variables insertion transformation. The number of the data cache misses is increased slightly due to the dummy variables insertion (because it may increase the

number of the compulsory misses).

Table 5-21 The result of Exp#5.2

Parameters	Original	Transformed	%
Code Size (bytes)	304	300	-1.32
Instruction Cache Misses	11	10	-9.09
Data Cache Misses	109	110	0.92
CPU Cycles	20689	20609	-2.13
Energy Consumption (nJ)	37316	37191	-1.88

From Figure 5-7, we find that it needs more than one instruction to calculate the base address of the array ‘a’ before transformation. After transformation, it only needs one instruction to calculate the bases address of the every array. Hence, the CPU cycles consumed are reduced resulting in the reduced energy consumption. Because $\forall E_{BAi}' = E_{BA}$, according to Eq. (5.19), the energy consumption savings can be derived as:

$$\Delta E \approx \sum_{i=0}^{N_a-1} [N_{BAi} \times (E_{BA} - E_{BAi})] + \left[\frac{\sum_{i=0}^{N_e-1} (S_{BAi}' - S_{BAi})}{S_{ICL}} \right] \times E_{ICM} + (N_{DCM}' - N_{DCM}) \times E_{DCM} \quad \text{Eq. (5.21)}$$

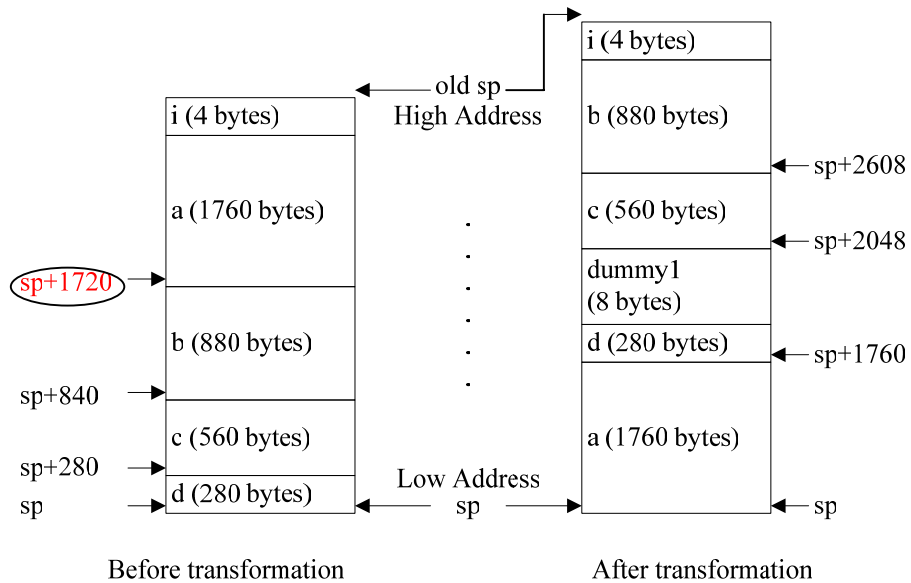


Figure 5-7 The stack content of Exp#5.2

Limitation: It is proposed according to ARM ISA. Besides, it is strongly tied to the strategies of the compiler to calculate the base addresses of the arrays.

5.5.3 Arrays Declaration Permutation

Assumption: the modified data structures will not result in changes in the number of the conflict misses.

As shown in Table 5-22, the code size and the number of the instruction cache misses are reduced after arrays declaration permutation transformation. The number of the data cache misses is not affected.

Table 5-22 The result of Exp#5.3

Parameters	Original	Transformed	%
Code Size (bytes)	304	300	-1.32
Instruction Cache Misses	11	10	-9.09
Data Cache Misses	109	109	0.00
CPU Cycles	20689	20233	-2.20
Energy Consumption (nJ)	37316	36567	-2.01

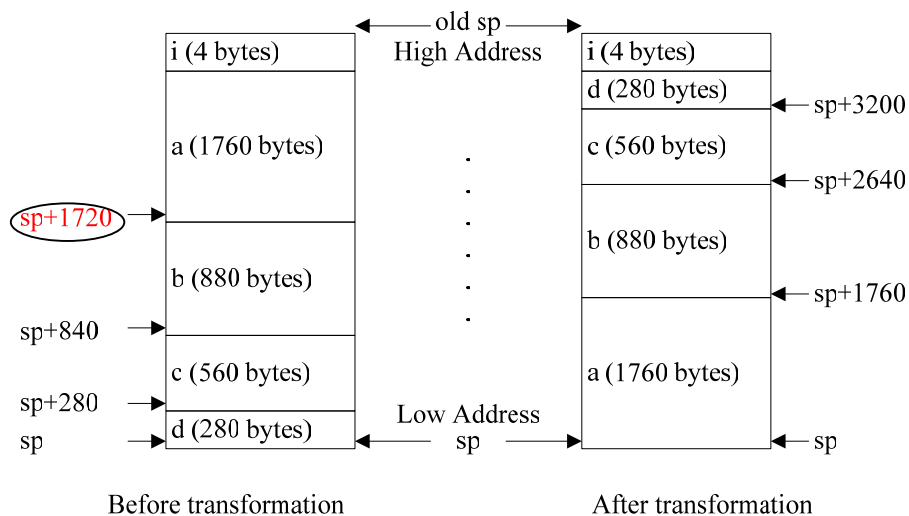


Figure 5-8 The stack content of Exp#5.3

Figure 5-8 shows the stack content of Exp#5.3. The goal of this transformation is the same as dummy variables insertion one. It tries to reduce the number of the instructions executed to calculate the bases address of the every array. Hence, the CPU cycles consumed are reduced resulting in the reduced energy consumption. When this transformation can get the best solution, $\forall E_{BAi}' = E_{BAi}$. And because N_{DCM}' is equal to N_{DCM} , according to Eq. (5.19), the energy consumption savings can be derived as:

$$\Delta E \approx \sum_{i=0}^{N_a-1} [N_{BAi} \times (E_{BA} - E_{BAi})] + \left[\frac{\sum_{i=0}^{N_a-1} (S_{BAi}' - S_{BAi})}{S_{ICL}} \right] \times E_{ICM} \quad \text{Eq. (5.22)}$$

Limitation: It is proposed according to ARM ISA. Besides, it is strongly tied to the strategies of the compiler to calculate the base addresses of the arrays.



5.6 Summary

Table 5-23 The results after transformation in our experiments

Transformation	Exp. No.	Code Size	Performance	Energy Consumption
Common sub-expression elimination	Exp#1.1	↓ 5.00%	↑ 22.08%	↓ 17.92%
Loop fusion	Exp#2.1	↓ 29.55%	↑ 30.60%	↓ 28.78%
Loop fission	Exp#2.2	↑ 0.32%	↑ 48.34%	↓ 62.76%
Loop reversal	Exp#2.3	→	↑ 6.02%	↓ 5.22%
Loop inversion	Exp#2.4	→	↑ 0.55%	↓ 1.77%
Loop interchange	Exp#2.5.a	→	↓ 0.69%*	↑ 0.58%*
	Exp#2.5.b	→	↑ 38.29%	↓ 53.97%
Loop unrolling (at best unrolling factor)	Exp#2.6	↑ 534.38%	↑ 25.48%	↓ 22.96%
Loop unswitching	Exp#2.7.a	→	↑ 98.10%	↓ 96.95%
	Exp#2.7.b	↑ 31.25%	↑ 23.56%	↓ 21.98%
Conditional sub-expression reordering	Exp#3.1	→	↑ 60.13%	↓ 59.15%
Procedure inlining	Exp#4.1	↑ 71.03%	↑ 7.65%	↓ 5.62%
Procedure integration	Exp#4.2	↑ 26.17%	↑ 8.07%	↓ 6.58%
Loop embedding	Exp#4.3	→	↑ 28.92%	↓ 27.36%
Arrays declaration sorting	Exp#5.1.a	↑ 3.57%	↓ 2.30%	↑ 2.04%
	Exp#5.1.b	↑ 1.32%	↑ 0.39%	↓ 0.33%
Dummy variables insertion	Exp#5.2	↓ 1.32%	↑ 2.13%	↓ 1.88%
Arrays declaration permutation	Exp#5.3	↓ 1.32%	↑ 2.20%	↓ 2.01%

(↓: fall, ↑: rise, →: unchanged or almost unchanged)

*: impacted by ARM ISA

As a summary, Table 5-23 lists the results after transformation in our experiments. From the experimental results of the transformations, the expected results of the energy consumption and the side effects are listed in Table 5-24. It is noted that loop fusion, loop fission, conditional sub-expression reordering, dummy variables insertion and arrays declaration permutation are under the assumptions which are listed in their related sections. And loop reversal, arrays declarations sorting, dummy variables insertion and arrays declaration permutation are expected to reduce the energy consumption when the limitations

which are listed in their related sections are met. Besides, we also find that loop interchange and loop reversal are impacted by ARM ISA.

Table 5-24 The expected results after transformation

Transformation	Code Size	Performance	Energy Consumption
Common sub-expression elimination	↓	↑	↓
Loop fusion	↓	↑	↓
Loop fission	↑	↑	↓
Loop reversal	→	→ or ↑*	→ or ↓*
Loop inversion	→	↑	↓
Loop interchange	→	↓* or ↑	↑* or ↓
Loop unrolling (at best unrolling factor)	↑	↑	↓
Loop unswitching	→ or ↑	↑	↓
Conditional sub-expression reordering	→	↑	↓
Procedure inlining	↑	↑ or ↓	↓ or ↑
Procedure integration	↑	↑	↓
Loop embedding	→	↑	↓
Arrays declaration sorting	-	-	-
Dummy variables insertion	↓	↑	↓
Arrays declaration permutation	↓	↑	↓

(↓: fall, ↑: rise, →: unchanged or almost unchanged, - : uncertain)

*: impacted by ARM ISA

Chapter 6

Conclusion and Future work

We present an experiment framework to evaluate and analyze the impacts of the transformations in source code level on energy consumption. A series of transformations have been verified for both their energy-efficiency and their side effects on our target architecture. From the experimental results, we find that loop interchange and arrays declaration sorting transformations are impacted by the ISA. Based on ARM ISA, two transformations which include dummy variables insertion and arrays declaration permutation are proposed. It is expected that the two transformations will reduce energy consumption by reducing the number of the instructions executed to calculate the base addresses of arrays. Their experimental results are also the same as expected.

Thus far, a series of transformations are collected and verified their energy-efficiency. In the future, we can write our SUIF passes to implement the transformations. Besides, it is a critical topic to solve the phase-order problem for us to implement an automatic tool. The automatic tool is very useful to do transformations on source code files automatically and to generate better energy-efficient ones under some constraint, such as code size and performance.

References

- [1] V. Tiwari, S. Malik, and A. Wolfe, "Power Analysis of Embedded Software: A First Step towards Software Power Minimization," *IEEE Transactions on VLSI Systems*, Vol. 2, No.4, pp. 437-445, December 1994.
- [2] J. T. Russell and M. F. Jacome, "Software Power Estimation and Optimization for High Performance, 32-bit Embedded Processors," in *Proc. International Conference on Computer Design: VLIS in Computers and Processors*, October 1998, pp. 328-333.
- [3] S. Nikolaidis, T. Laopoulos, and A. Chatzigeorgiou, "Developing an Environment for Embedded Software Energy Estimation," in *Proc. Second IEEE International Workshop on Intelligent Data Acquisition and Advanced Computing Systems: Technology and Applications*, September 2003, pp. 20-24.
- [4] Y. Li and J. Henkel, "A Framework for Estimating and Minimizing Energy Dissipation of Embedded HW/SW Systems," in *Proc. 35th Design Automation Conference*, June 1998, pp. 188-193.
- [5] T. Simunic, L. Benini, and G. De Micheli, "Energy-Efficient Design of Battery-Powered Embedded Systems," *IEEE Transactions on Very Large Scale Integration Systems*, Vol. 9, No. 1, pp. 15-28, February 2001.
- [6] T. K. Tan, A. Raghunathan, and N. K. Jha, "A Simulation Framework for Energy-Consumption Analysis of OS-Driven Embedded Applications," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 22, No. 9, pp. 1284-1294, September 2003.
- [7] E. Y. Chung, L. Benini, and G. De Micheli, "Source Code Transformation based on Software Cost Analysis," in *Proc. 14th International Symposium on System Synthesis*, September-October 2001, pp. 153-158.

- [8] Y. Fei, S. Ravi, A. Raghunathan, and N. K. Jha, "Energy-Optimizing Source Code Transformations for OS-driven Embedded Software," in *Proc. 17th International Conference on VLSI Design*, January 2004, pp. 261-266.
- [9] T. K. Tan, A. Raghunathan, and N. K. Jha, "Software Architectural Transformations: A New Approach to Low Energy Embedded Software," in *Proc. Design, Automation and Test in Europe Conference and Exhibition*, March 2003, pp. 1046-1051.
- [10] H. Mehta, R. M. Owens, M. J. Irwin, R. Chen, and D. Ghosh, "Techniques for Low Energy Software," in *Proc. International Symposium on Low Power Electronics and Design*, 1997, pp. 72-75.
- [11] T. K. Tan, A. Raghunathan, and N. K. Jha, "Embedded Operating System Energy Analysis and Macro-modeling," in *Proc. 2002 IEEE International Conference on Computer Design: VLSI in computers and Processors*, September 2002, pp. 515-522.
- [12] A. Peymandoust, T. Simunic, and G. De Micheli, "Low Power Embedded Software Optimization using Symbolic Algebra," in *Proc. Design, Automation and Test in Europe Conference and Exhibition*, March 2002, pp. 1052-1058.
- [13] S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers Inc, August 1997.
- [14] R. Morgan, *Building an Optimizing Compiler*. Digital Press, 1998.
- [15] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto, "The Impact of Source Code Transformations on Software Power and Energy Consumption," *Journal of Circuits, Systems and Computers*, Vol. 11, No. 5, pp. 477-502, May 2002.
- [16] C. Brandolese, W. Fornaciari, F. Salice, and D. Sciuto, "Analysis and Modeling of Energy Reducing Source Code Transformations," in *Proc. Design, Automation and Test in Europe Conference and Exhibition Designers' Forum*, February 2004, pp. 306-311.
- [17] E. Y. Chung, G. De Micheli, M. Carilli, L. Benini, and G. Luculli, "Value-base Source Code Specialization for Energy Reduction," *ST Journal of System Research*, Vol. 3, No.

- 1, pp. 29-48, April 2002.
- [18] L. Benini and G. De Micheli, "System-Level Power Optimization: Techniques and Tools," *ACM Transactions on Design Automation of Electronic Systems*, Vol. 5, No. 2, pp. 115-192, April 2000.
- [19] M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Compiler Optimizations for Low Power Systems," in *Power Aware Computing*, pp. 191-210. Kluwer Academic Publishers, Jun 2002.
- [20] W. Zhang, J. S. Hu, V. Degalahal, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Reducing Instruction Cache Energy Consumption Using a Compiler-Based Strategy," *ACM Transactions on Architecture and Code Optimization*, Vol. 1, No. 1, pp. 3-33, March 2004.
- [21] S. V. Gheorghita, H. Corporaal, and T. Basten, "Iterative Compilation for Energy Reduction," *Journal of Embedded Computing*, Vol. 1, No. 4, pp. 509-520, December 2005.
- [22] K. S. McKinley, S. Carr, and C. W. Tseng, "Improving Data Locality with Loop Transformations," *ACM Transactions on Programming Languages and Systems*, Vol. 18, No. 4, pp. 424-453, July 1996.
- [23] P. R. Panda, H. Nakamura, N. D. Dutt, and A. Nicolau, "Augmenting Loop Tiling with Data Alignment for Improved Cache Performance," *IEEE Transaction on Computers*, Vol. 48, No. 2, pp. 142-149, February 1999.
- [24] M. E. Lee, "Optimization of Computer Programs in C," April 1999. [Online]. Available: <http://leto.net/docs/C-optimization.php>
- [25] *Application Note 34: Writing Efficient C for ARM*, Advanced RISC Machine Ltd, January 1998. [Online]. Available: http://www.arm.com/pdfs/DAI0034A_efficient_c.pdf
- [26] M. Boehhold, I. Karkowski, and H. Corporaal, "Transforming and Parallelizing ANSI C Programs using Pattern Recognition," in *Proc. 7th International Conference on High*

Performance Computing and Networking, April 1999, pp. 673-682.

[27] D. Seal, *ARM Architecture Reference Manual*. Addison-Wesley, December 2000.

[Online]. Available: <http://www.arm.com/community/university/eulaarmarm.html>

[28] ARM Procedure Call Standard. [Online]. Available:

<http://www.chiark.greenend.org.uk/~theom/riscos/docs/CodeStds/APCS.txt>

[29] *Procedure Call Standard for the ARM Architecture*, Development System Division,

Compiler Tools Group, January 2007. [Online]. Available:

<http://www.arm.com/pdfs/aapcs.pdf>

[30] GCC 2.95.3 Manual. [Online]. Available:

<http://gcc.gnu.org/onlinedocs/gcc-2.95.3/gcc.html>

[31] Embedded StrongARM Energy Simulator (EMSIM-2.0): Year 2003. Available:

<http://www.princeton.edu/~cad/emsim/>

[32] T. K. Tan, A. Raghunathan, and N. K. Jha, "EMSIM: An Energy Simulation Framework for an Embedded Operating System," in *Proc. International Symposium on Circuits and Systems*, May 2002, pp. 464-467.

[33] *Intel StrongARM SA-1100 Microprocessor Developer's Manual*, Intel Corporation,

August 1999. [Online]. Available: <http://www.lartmaker.nl/278088.pdf>

[34] The Stanford SUIF Compiler Group. Available: <http://suif.stanford.edu>

[35] G. Aigner, A. Diwan, D. L. Heine, M. S. Lam, D. L. Moore, B. R. Murphy, and C.

Sapuntzakis, "An Overview of the SUIF2 compiler Infrastructure," Computer Systems Laboratory in Stanford University and Portland Group, Inc. [Online]. Available:

<http://suif.stanford.edu/suif/suif2/doc-2.2.0-4/overview.ps>

[36] ExPress Group. Available: <http://express.ece.ucsb.edu/index.html>

[37] BRASS Research Group. Available: <http://brass.cs.berkeley.edu/index.html>

[38] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*.

MIT Press, 2001.

[39] Alexander Bogomolny's Permutations Web Page. Available:

http://www.cut-the-knot.org/do_you_know/AllPerm.shtml

[40] M. D. Hill and A. J. Smith, "Evaluating Associativity in CPU Caches," *IEEE*

Transactions on computers, Vol. 38, No. 12, pp. 1612-1630, December 1989.

