國立東華大學資訊工程學系

碩士論文

Procedural Abstraction 應用在程式最佳化的研究

A Code Size Optimization Using Procedural Abstraction

研 究 生：陳 信 霖

指導教授：雍忠 博士

中華民國九十二年七月

A Code Size Optimization Using Procedural Abstraction

by

Hsin-Lin Chen

A thesis submitted in partial fulfillment

of the requirements for the degree of

Master of Science

Department of Computer Science and Information Engineering

National Dong Hwa University

July 2003

Approved: ____________________

Chung Yung

© Hsin-Lin Chen

All Rights Reserved, 2003

To my memory father, dear mother, Fancy Su, and Lulu Su

Acknowledgements

I finally complete this thesis through three years of school day. Because I am an on-the-job graduate student, I must proceed my school work, meanwhile, I keep at my job. Therefore, I ascribe accomplishing smoothly the thesis to the support and cheer given by many people. I would like to express my wholehearted thankfulness for their encouragement during the period of school day.

My advisor, Professor Chung Yung, I would communicate my thank to him first of all. He provides a guide of the method and manner of researching by continuous discussions and teaching for me, of course it is including working manner and studying viewpoint himself. It is a different experience from my career. I also thank the thesis committee members, Dr. Tyng-Ruey Chung and Dr. Shi-Ching Yen, for their constructive commentaries on this thesis.

In addition, there are many people who gave deep helps to me during my school age, and I would like to express my acknowledgement to them, my laboratory members especially. They are Hsun-He Wang, Yu-Ching Lan, Ming-Hao Jiang, Li-Bang Chen, Hsiang-Wei Kong, Feng-Wei Yang, Ming-Hsien Lin, Yun-Long Yu, Yu-Liang Sun, Jih-Jin Ye, Chih-Chiang Li, and Bo-

Ren Chuang. Finally, I would like to thank my professors, friends, classmates, and colleagues who once gave some helps to me. Because I had been given their supports and encouragements, I have a varied and wonderful graduate student-lift.

摘要

近年來嵌入式系統(Embedded Systems)發展迅速，其所提供的功能已越來越複雜，使得嵌入式系統的系統資源(System Resources)需求相對提高。由於嵌入式系統的特性，其系統資源並不能無限制的擴充，所以降低嵌入式系統應用軟體的程式大小(Object code size)，可以有效的提昇嵌入式系統資源的使用效能。

其中 Procedural Abstraction 是一個有效降低程式大小的方法，此方法通常應用在低階(Low-Level stage)程式語言，譬如 assembly 或是 machine code。現在,越來越多嵌入式系統使用高階語言發展應用軟體，例如 C、C++、Java。所以，我們思考 procedural abstraction 是否可以應用在高階語言，比應用在低階語言更能降低程式大小。

在這篇論文，我們研究是否有一機制可以來判斷一段程式在轉換成 procedural abstraction 之後是否能降低程式大小。我們提出 model 稱為 procedural abstraction criteria，並同時討論 procedural abstraction 應用在低階及高階語言，觀察目的程式大小的變化情形。為了討論高階語言的 procedural abstraction 的需要，我們實作新的編譯器，稱為 C-to-Thumb Compiler,包含 C parser 以及 Thumb Code Generator。在這篇論文，我們針對低階語言以及高階語言分別實作些實驗，並對這些實驗值，討論 procedural abstraction criteria 對程式大小的影響。

A Code Size Optimization Using Procedural Abstraction

Hsin-Lin Chen

Advisor: Chung Yung

Abstract

As the functions of an embedded system become more and more complex, the resources demanded by an embedded system increase by a lot in recent years, although the resources are always limited in practice. One of the effective methods proposed to reduce the code size is procedural abstraction, which is usually applied at the low level stage, such as the assembly or machine code stage. Since more and more applications for embedded systems are developed in high-level languages like C, C++, and Java, it is a natural way of reducing the size of object code using procedural abstraction at the high level stage. In this thesis, we propose a model of deciding whether a procedural abstraction transformation reduces the size of object code. We present the procedural abstraction criteria at the low level stage and at the high level stage in order for the procedural abstraction transformation to effectively reduce the size of object code. We design a new compiler optimization based on the criteria such that the optimized object code is guaranteed to have a reduced size. We implement the proposed optimization in a simple C-to-Thumb compiler. This thesis also includes the benchmarks of the experiments on our new optimization.

Contents

Dedication **iii**

Acknowledgements **iv**

List of Figures **xi**

1 Introduction **1**

1.1 Background 3

1.2 Reducing the Resources Required 5

1.2.1 Recycling the Unused Data Memory 5

1.2.2 Code Compression 6

1.2.3 Code Compaction 11

1.3 Motivation 12

1.4 Methodology 16

1.4.1 Methodology of the Low-Level Stage 16

1.4.2 Methodology of the High-Level Stage 17

1.5 Thesis Organization 19

2 Related Work **20**

2.1 Traditional Optimization for Code Compression 20
2.1.1 Unreachable Code Elimination 20
2.1.2 Strength Reduction 22
2.1.3 Dead Code Elimination 23
2.1.4 Redundant Code Elimination 24
2.2 Procedural Abstraction 25
2.2.1 Code Factoring 26
2.2.2 Cross-Jumping 27
2.2.3 Procedural Abstraction 28
2.2.4 Identify the Repeated Code 29
3 Procedural Abstraction at Low-Level Stage 32
3.1 Procedural Abstraction Criteria at the Low-Level Stage 33
3.2 Experiments 35
3.3 Procedural Abstraction Criteria 39
3.4 Procedural Abstraction Model 40
4 Procedural Abstraction Criteria at the High-Level Stage 43
4.1 Procedural Abstraction Criteria at the High-Level Stage . . . 45
4.2 The C-to-Thumb Compiler 48
4.2.1 Compiler Framework 48
4.2.2 Compiler Developmental Environment 51
4.2.3 Compiler Specification 52
4.2.4 An Example 55

4.3 Experiments 55

4.4 Procedural Abstraction Criteria 56

4.5 Procedural Abstraction Criteria Model at the High-Level Stage 57

5 Conclusions 66

List of Figures

1.1 The basic concept code compaction and code compression . . 2

1.2 The example of uncompressed assembly code 13

1.3 The example of compressed assembly code 14

2.1 The example of unreachable ode 21

2.2 The example of Dead code elimination 24

2.3 The example of Redundant code elimination 25

2.4 The example of Code-Factoring 27

2.5 The example of cross-jumping 28

2.6 The example of Cross-jumping applied to control flow graph . 29

2.7 The example of suffix tree. 31

3.1 A code fragment: (a) the inline style and (b) the procedural abstracted style 33

3.2 The ARM Project Manager 37

3.3 The ARM Debugger for Windows 38

3.4 Benchmarks from the experiments on the procedural abstraction criteria at the low level stage 38

3.5 The algorithm for the procedural abstraction criteria at the low level stage 41

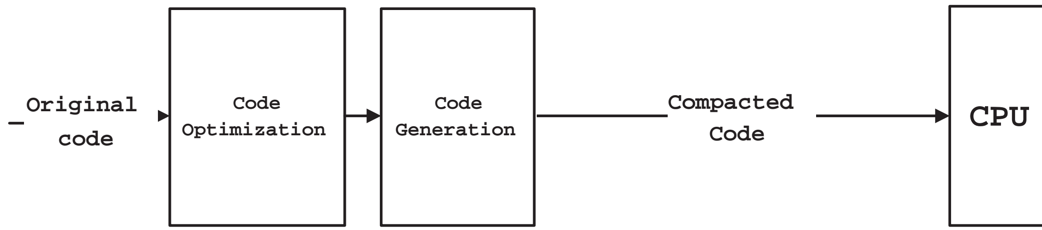
4.1 The structure diagram of applying procedural abstraction: (a) tree-based approach (b) conventional approach 44
4.2 A simple program in C (a) the inline style (b) the procedural abstracted style 45
4.3 The examples of the inline style and the procedural abstracted style 46
4.4 The structure diagram of applying procedural abstraction at the high level stage 47
4.5 The process to construct a C-to-Thumb Compiler. 49
4.6 Using the C-to-Thumb Compiler to compile the C source program. 50
4.7 Using the C-to-Thumb Compiler (Snapshot 1). 53
4.8 Using the C-to-Thumb Compiler (Snapshot 2). 53
4.9 Using the C-to-Thumb Compiler (the output of the compiler). 54
4.10 Benchmarks from the experiments on the procedural abstraction criteria at the high level stage 56
4.11 The algorithm for the procedural abstraction criteria at the high level stage 58
4.12 The algorithm for estimate the high-level Procedural abstraction criteria. 62
4.13 The algorithm to compute the code size of the single high-level statement. 63

4.14 Benchmarks from the experiments on the PA criteria algorithm. 65

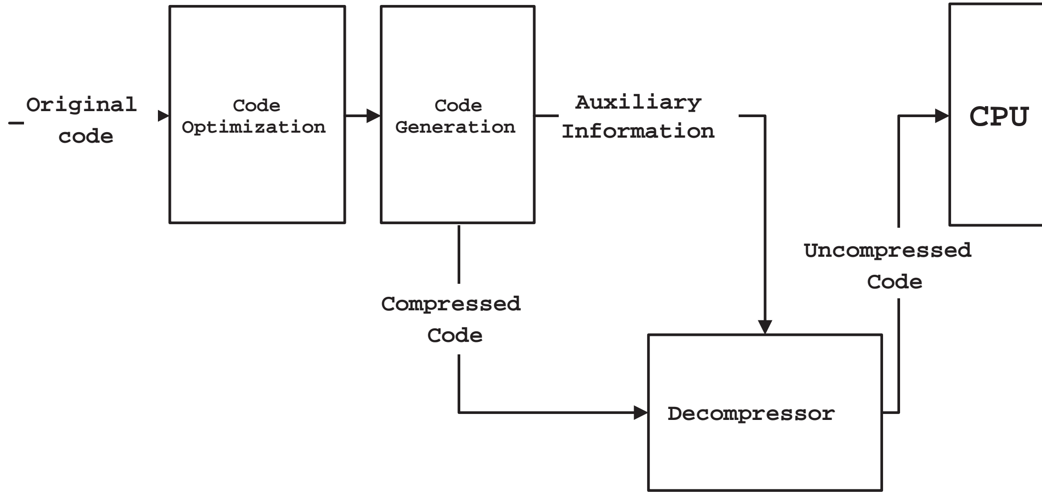
Chapter 1

Introduction

The code compression technique is a wide-range topic discussed in various fields of computer science, such as compilers, code optimization, etc. According to the ways of generating compressed codes with the functionality preserved, the proposed approaches can be grouped into two types. Some of the approaches generate object codes that can be executed directly, and hence they do not need decompression before executing the programs. We call this way as *code compaction* [DEMS00], showed in Figure 1.1(a). For the other approaches, the compressed codes generated by compilers must be decompressed before executing. This way is called *code compression*, showed in Figure 1.1(b). In this thesis, we focus on code compaction and its application to the code optimization of compilers. Our research destined for the embedded systems because of the characteristics that the system resources such as memories, caches, and processors are limited, which is suitable for code compression to reduce the system resource requirements[YSOOS97]. In the optimization topics, procedu-



(a)A code compaction diagram



(b)A code compression diagram

Figure 1.1: The basic concept code compaction and code compression

ral abstraction is a commonly used method to generate optimal object codes. However, it is usually applied at the low-level stage like assembly language. We will extend the research range from the low-level stage to the high-level stage in this thesis. In this chapter, we will introduce the background concept and research motivation related with embedded systems, code compression, and procedural abstraction.

1.1 Background

In recent years, due to the progress on the development techniques of semiconductor manufacturing, and the integrity on the communication infrastructure, and furthermore, the increasing demands for personal communication and mobil computing, the development of embedded systems is very fast and mature, in both hardware and software. To combine hardware, operating system, and kernel application software with a single chip completing all functions is called *System on Chip(SoC)*. The SoC technique is a current trend to develop modern embedded system devices. The application for embedded systems includes wireless communications, information electric equipments, personal communicators, industrial automatic facilities and scientific instruments, and so on. The characteristics of these services are minisized, regional, functional, personal, facilitate, etc. As a result, the demands for embedded system devices become larger and larger. So that, the techniques of embedded systems are a key to the developments of the electronic equipments in the future.

The microprocessors for the embedded systems can be built very small by the progress on the capacity of the transistors in a single chip. Now, they include functions as powerful as these on desktop systems. The faster the embedded systems develop, the more services that an embedded system may support. For example, a general-propose of personal digital assistant(PDA) can support various services including the personal data processing, multimedia services, graphic processing, telecom, LAN, wireless-LAN, voices, and external I/O devices controlling, etc. A single embedded system can sup-

ported more and more services for a variety of areas even though some of the embedded systems are designed for specialized areas. Therefore, a modern embedded system must be able to execute dynamically various application programs to handle complex requirements. Relatively, the software on the embedded system becomes more complex. Once the application software on the embedded systems is more and more complex, their demands for the system resources, including memories, local storages, remote storages via network, processor caches, processor computing, and consumption of energy resources, are growing fast. It seems unlikely that the hardware resources of the embedded systems expanded unlimitedly. It is much slower than expansion on application software. It is important to make effective use of system resources. If the embedded systems use the hardware and software resources effectively, it can improve the overall performance and reduce the system resources required. Therefore, the purpose of this thesis is using the code optimization techniques to reduce the software complexity for the purpose of reducing both software and hardware consumption.

In general, each software program statement can generate several of the low-level statements (machine code) by the code generator of a compiler. So that, a program without the process of compiler code optimizations probably would produce a machine code with a larger size than the optimal code. If the program code size is larger, the execution of the program require more resources. We say that if we can reduce the size of the source code programs effectively, we can certainly improve the system performance. In this thesis,

we focus on the topic how to reduce the code sizes in order to reduce the resources consumption for improving the system performance.

1.2 Reducing the Resources Required

As usual, the growth of the system resources is much slower than the growth of the need. For this reason, using system resources effectively can further improve the system performance, and reduce the system resources required [Cam88]. There are three popular methods to reduce the use of system resources:

1. *Recycling the unused data memory*,
2. *Code compression*, and
3. *Code compaction*.

1.2.1 Recycling the Unused Data Memory

The system reduces the memory used by recycling the unused data memory as soon as possible at run time [LW99]. However, the overhead may be very high in this way. If a processor has only limited amount of data memories at run time, it may result in more time consumed at CPU processing and swapping between caches and memories, and hence the overall performance is down.

When the system has more and more running processes concurrently, there are less and less system resources available at run time. It causes the operating system to spend a lot of time to swap the system resources that processes need.

Some of the operating systems include some advanced operating system modules [SB94, ASU86] such as scheduling functions, and memory-management functions to reduce the system loading, but it still needs to spend the process-times to assign and swap the system resources because the processor cannot load with all the required resources into main memory at execution time. These kinds of operating system usually are powerful, and they are not suitable for the embedded systems. For an embedded system, it can not effectively improve the system performance by reducing the memory used with recycling the unused data memory.

1.2.2 Code Compression

The second method is to make use of the data compression techniques in order to reach reducing the object code size, as shown in Figure 1.1(b). The source code for compression in this way is usually the low-level programs such as machine codes and assembly codes. The low-level languages have fixed format and fixed length of instruction[LSC02]. For example, the code sizes of the machine instructions are usually fixed according to the addressing modes that related to the addresses-bus sizes and the size of data buses the embedded micro-processors.

In general, the size of machine code is a multiple of the size of *CPU-WORDs*. The instruction formats of assembly language are also fixed;[ARM00a, ARM98a, ARM98b, ARM00b]. The follow instruction formats and examples are extracted from *ARM Architecture Reference Manual* [ARM00b].

1. *[operator,operand].*

   Example : $PUSH < reglist >$; Push registers onto stack.

2. *[operator,operand1,operand2].*

   Example : *ADD Rd, Rm* ; Rd := Rd + Rm .

3. *[operator,operand1,operand2,operand3].*

   Example : *ADD Rd, Rn, Rm* ; Rd := Rn + Rm.

For the low-level languages, we can take the instructions as formal data and use the traditional data compression techniques to compress the programs. We can use the *compression-ratio* to measure the compression degree. The compression ratio in this way is estimated as good [ACCP98].

The compression-ratio is defined as follow. If compression-ratio is smaller, it is expressed to get a better compressed effect.

$$compression - ratio = \frac{compressed\ code}{original\ code}$$

Data compression is applied to many areas in computer science. For example, the data compression can make use of data communications, image processing, data encoding, cryptography, and text compression, etc. There are many researches and implementations for data compression [LH87]. The methods of data compression, can be separated into two kinds according to whether the compression techniques have relations with data or not. The first kind of the techniques is called *Content-Dependent Technique* which has relations with data themselves [Hol95]. The methods of first kind include Compression of

Repeated Characters [Ant97], Dictionary Substitution [Hol95], Compression of Records of Data, and Differencing Coding, etc. The other kind is called *Content-Independent Techniques*, for example Huffman's Encoding [Duf52], Run Length Encoding [RR01], Arithmetic Encoding [Ant97, WNC87], and Lempel-Ziv-Welch Encoding (LZW Encoding) [BWC89]. The data compression techniques are also popularly applied to the code compression.

The related work on code compression may be divided into three different levels: the high (source code) level, the medium (IR, or byte code) level, and the low (assembly, or machine code) level [DEM99].

- The high level: The approaches in high-level code compression analyze the program from the source code or the tokens [Eva98]. Cameron uses *the syntactic information source models* [Cam88] for source encoding and data encoding. The syntactic models characterize messages in terms of their grammatical structures, in contrast to the lexical models dealing with the character-by-character composition of messages. This approach is implemented for Pascal programs, and its compression ratio is around 20% [Cam88].

- The medium level: The approaches in medium-level code compression analyze the program from the intermediate representation (IR) generated by the compiler front-end. Usually, the formats of IR represent the data structures used in the compiler. So that, general algorithms may be applied to analyze the program and implement the compression. The compression ratio of the approaches in this level is ranged from 20% to

40% [EF01, Luc00].

- The low level: The approaches in low-level compression analyze the program from the machine code or the assembly code. There are a few approaches focused on this level [DEMS00, Luc00, BZIP2h, GZIPh]. In the low level, the format is usually fixed, and hence several popular compression methods may be used, such as pattern matching, and Huffman encoding [Duf52]. Generally, the compression ratio of the approaches in this level is high, ranged form 30% to 70%[DEMS00, EF01, Luc00, BZIP2h].

Most of the code compression methods fall into two general types [LBCM97] :

1. Statistical compression, and
2. Dictionary compression.

Statistical compression [Fra99] determines the size of the codeword by the frequency of a single word. It replaces the original words with the codewords. If the words appear more frequently, they are encoded into shorter codewords, so that the sizes of the compressed code are minimum. The Huffman Encoding [Duf52] is a well-known method of statistical compression.

Dictionary compression [ACAP00] makes use of an algorithm to determine the codewords that is used to replace the original code. A dictionary compression constructs some auxiliary data like dictionary structures or table structures to record the information about every codeword as a dictionary entry. Every entry of the dictionary contains two items of a codeword; namely,

an index to the original place, and the original characters it compresses.

Dictionary compression can compress the programs effectively because the codewords use fewer characters than the original codes. The compression ratio of this method is as follow:

$$compression - ratio = \frac{compressed\ code + auxiliary\ data}{original\ code}$$

The code generator of a compiler may generate the compressed codes with a number of auxiliary data at the back-end stage. The compiler reads the source program, and generates the compressed codes with the auxiliary data at compile time [WC92, LBCM97]. At run time, the loader sends the compressed codes together with the auxiliary data to the processor. The processor decompresses the codes according to the dictionary and replaces with the original codes before executing it. The auxiliary data is usually the data structure for the code dictionary. It provides the information about how the compressed code may be decompressed. The approaches of this method use several traditional data compression algorithms, such as Huffman encoding [EEFLP97, Duf52]. With this approach, the compressed codes are effectively small, but the overhead is relatively high at the decompression stage. At run time, it needs to read the compressed codes and the dictionary entries as input, and generates uncompressed codes before executing. Therefore, system must consider the time consumed in decompressing time and processing.

1.2.3 Code Compaction

The third type of code compression is making use of traditional compiler techniques. The classical compiler optimizations can effectively reach the good of reducing the code size as code compression. The compiler front-end generates the *intermediate representation* (IR) as input and transferred them into the optimization module that optimizes the IR codes before generating the executable codes by compiler back-end. Now, the optimizing compiler can effectively generate minimal size of executable codes.

The optimization mentioned here is not a single program to cover all functions. It is a combination of a few individual optimization programs for each function. A compiler can choose the optimization functions for its use. For example, for the demand of minimal code size or the demand of minimal execution time, or both, a compiler may have different optimization subroutines to achieve the demand. Some of the optimizations applied to reduce are listed as follows[Muc97].

1. Elimination of redundant,
2. Elimination of unreachable code,
3. Elimination of dead code,
4. Strength reduction,
5. Code factoring, and
6. Procedural abstraction.

We will discuss them in details in at chapter 2.

The code generator of a compiler generates a special form of compressed code called a *compacted form*. Different from the other methods, the compacted code is directly executable; i.e., this method does not need to compress and/or decompress the code. This method applies several techniques of compiler optimization to analyze the source program and generate a compacted code. It is usually referred to as a *zero-overhead* method [EEFLP97]. Since the method is usually included in an expanded compilation stage, the run time overhead is the smallest of the three. In this thesis, we focus on the investigation of this method.

1.3 Motivation

The compiler optimization has a commonly used method called procedural abstraction. It can reduce the object code size through the procedural abstraction before code generation stage.

Figure 1.2 is a fragment of a Thumb assembly code that run at a series ARM embedded processors. The instructions at from 11 to 14 do the following task. At first, processor loads the address of a variable c from memory into register $r2$. Then, it reads the data from the address stored in $r2$ into $r0$. The next, it stores the constant data #0 into $r0$. At last, the data stored in $r0$ is moved to memory whose address is stored in $r2$. Clear, the four instructions do the same task as a C statement $c = 0$;. In the Figure, the four instructions repeat for three times.

```
00  header
01          ADR r0, start + 1
02          BX r0
            .
            .
-----------------------------
11          ADR r2, c
12          STR r0, [r2]
13          MOV r0, #0
14          STR r0, [r2]
-----------------------------
            .
            .
-----------------------------
21          ADR r2, c
22          STR r0, [r2]
23          MOV r0, #0
24          STR r0, [r2]
-----------------------------
            .
            .
-----------------------------
31          ADR r2, c
32          STR r0, [r2]
33          MOV r0, #0
34          STR r0, [r2]
-----------------------------
            .
            .
98          MOV pc, lr
99          END
```

Figure 1.2: The example of uncompressed assembly code

```
00  header                                  ..
01          ADR r0, start + 1               ..
02          BX r0                    myProc
            .                               ADR r2, c
            .                               STR r0, [r2]
11          B myProc                        MOV r0, #0
            .                               STR r0, [r2]
            .                               MOV pc, lr
21          B myProcr                       ..
            .                               ..
            .
31          B myProcr
            .
            .
98          MOV pc, lr
99          END
```

Figure 1.3: The example of compressed assembly code

When applying dictionary compression method [LC02], the compiler use a single codeword to express the four instructions, and replaces the original code fragment with the codewords. The replaced code fragments are stored into a table structured called a dictionary table. When the processor gets the compressed code and dictionary data at run time, it decompresses the code according to the before executing it.

The procedural abstraction makes use of the concept of code factoring to optimize the object codes. The mainly idea is to abstract the identical parts in a program or a basic block into a single procedure, and replace the original instructions with the procedural-call instructions. The Figure 1.3 shows the assembly code and its processed codes by a procedural abstraction optimization. The result of compressed code can reduce the codes size considerably. If

the number of the repeated times is larger enough, the procedural abstraction can get a smaller code size.

However, procedural abstraction does not always result in good compression. As an example, if the code fragment only appears once in the program, the code size generated by procedural abstraction function may be greater than the original code size. Because of the overhead for transforming the original code into a procedure and the size of a function-call instruction are probably greater than the size of original code, procedural abstraction may actually increase the size. To avoid this situation, a compiler must determine whether procedural abstraction can reduce the object code size or not.

In most of the previous approaches, procedural abstraction is applied at the low-level stage, such as assembly code. The idea of this thesis is base on the observation that the procedural abstraction may be applied at high level, such as C programs. If a compiler can determine whether the high-level program statement sequence transformed with procedural abstraction can reduce the object code size, the procedural abstraction transformation can proceed at the early stage of compilation.

Accordingly, we are interested in two topics about procedural abstraction applied at compiler optimization:

1. For a code fragment, find the numbers of repetition that the procedural abstraction may reduce the object code size. For those case the procedural abstraction does not reduce the code size, the compiler may just keep the original code.

2. Develop a model for the high-level procedural abstraction, so that we may apply the model to the programs executed on embedded processors for reducing the code size.

Similar to the low-level stage, high-level procedural abstraction must decide the number of repeated times for getting a result whether the procedural abstraction can reduce the object code size at high-level stage or not.

1.4 Methodology

Our researches explore the procedural abstraction applying to the code optimizations of the embedded systems. At first our directions of the researches are the procedural abstraction applying to the low-level stage.

1.4.1 Methodology of the Low-Level Stage

The works include as follows:

1. We propose a model for procedural abstraction criteria at low-level stage.

2. We make a experiment to explain the criteria model : If a assembly code fragment can be found a criteria number and the code size of the candidate fragment, we can decide whether transforming the assembly codes into PA mode may reduce the code size at low-level stage.

The first work is defining a framework or model for procedural abstraction at low-level stage, and finding out the procedural abstraction criteria in order to provide a compiler to generate the smaller object codes. When we have a

assembly code fragment, and the compiler want to know that which repeated time suitable for transforming the code fragment into PA mode. The model provides the information for the compiler.

We implement the experimental assembly codes to verify our framework according to the proposed low-level stage model. We use the ARM/Thumb assembly language to implement the procedural abstraction experiments. The experimental programs are written at both procedural abstraction mode and inline mode separately, and the programs are compiled using the ARM development tools.

We get the objects code sizes from the experiments after running at both procedural abstraction mode and inline mode. According to the experiment data, we can know that which repeated time and candidate code size can reduce the code size running at PA mode. Therefore, the proposed low-level criteria model are verified through the results of the experiments.

1.4.2 Methodology of the High-Level Stage

The next, we focus on the procedural abstraction applying to high level language. The researches include as follows:

1. We propose a model for procedural abstraction criteria at high-level stage.

2. We make a experiment to explain the criteria model : If the high-level statements can be found a criteria number and the code size of the candidate statements in a program, we can decide that transforming the

statements from the inline mode into the PA mode may reduce the code size.

3. For simplifying the relationship both the high-level statements and the low-level codes and implementing the experiments easily, we implement a experimental compiler called as C-to-Thumb compiler.

We extend the procedural abstraction concept from the low-level stage apply to the high-level stage. Similar to low-level stage, we also define a model to find out the *procedural abstraction criteria* at the high-level stage. Different from the low-level stage, the each high-level statement does not generate the same number of the assembly codes. Instead, the different high-level statements generate different number of assembly codes base on the complexity of the statements.

According the characteristics of discussing the high-level procedural abstraction, this thesis constructs an experimental C-to-Thumb compiler to verify our high-level procedural abstraction model. The C-to-Thumb compiler reads the C statements as input, and constructs the *intermediate representation* (IR) or *abstract syntax tree* (AST). Because our model works with the IR in the tree structures, the high-level procedural abstraction criteria model is also called as *tree-based procedural abstraction*.

To verify the model, we make the experiments for the high-level programs. The programs are written as both the PA mode and the inline mode respectively, and they have different repeated times for the candidate statements of the programs. We get the code sizes of the programs computed from the

experimental compiler. According to the experimental data of the programs, we will verify the model and find out which condition it may reduce the code size of the high-level statements.

1.5 Thesis Organization

We organize this thesis into several chapters. The first chapter gives a conceptual introduction of reducing the code size. We propose a problem of the procedural abstraction optimization applied to both the low-level and the high-level stage optimization. Chapter 2 is a brief introduction to procedural abstraction applied to code compression and the related researches. Chapter 3, we propose the procedural abstraction criteria at the low-level level stage. We describe the procedural abstraction model of deciding whether the procedural abstraction reduces the size of object codes.

We present the procedural abstraction criteria at the high level or parse-tree level stage in chapter 4. We construct a new compiler optimization and proceed the experiments on the high-level procedural abstraction criteria. We implement the optimization in a simple C-to-Thumb compiler that generates the ARM/Thumb assembly codes from the source programs in the ANSI C programming language. We will make a detailed description of the framework and implementation of the experimental compiler. We also describe our experiments and the benchmarks. At last is a brief conclusion and future direction.

Chapter 2

Related Work

The goal of the compiler optimizations is rather attend saving the execution time than reducing the code size. The focus of our research is how to save the system space and to reduce the code size on the embedded systems, on which the resources are limited. In this chapter, we introduce several related optimization approaches for reducing the code size.

2.1 Traditional Optimization for Code Compression

Traditional optimizations there are several effective methods for reducing the code size [DEMS00]. In this section, we discuss some important topics, such as dead code elimination, unreachable code elimination, etc.

2.1.1 Unreachable Code Elimination

If a code fragment or basic block never executes it, called *unreachable code*. Because of it has not any control flows entering to these code fragments [Muc97],

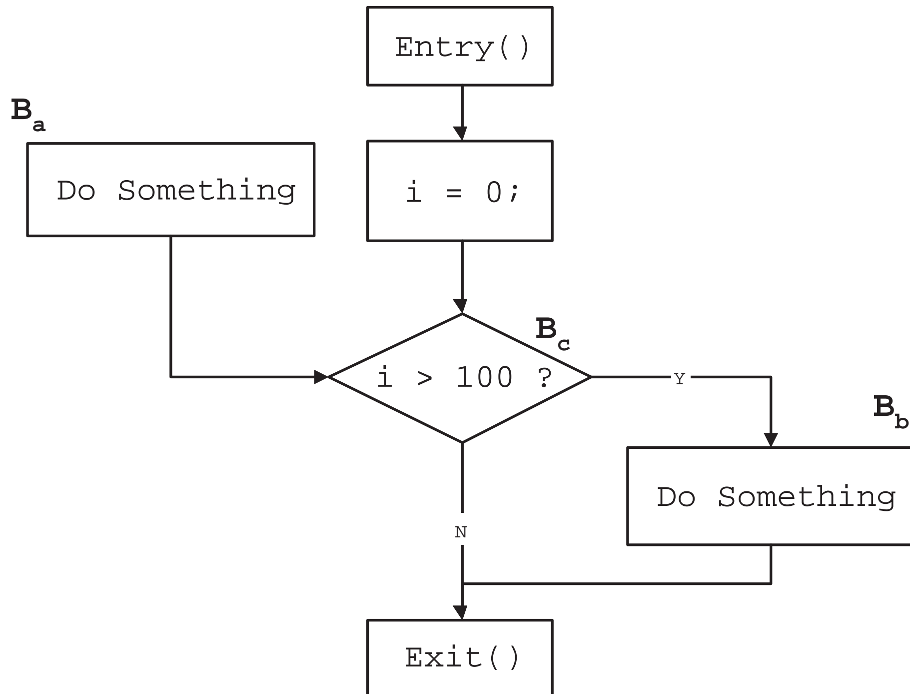


Figure 2.1: The example of unreachable ode

so that it has not any codes can possibly enter this basic block entry of the unreachable codes, however what are the input data and output data of the basic block. If the code fragment is unreachable code, it can eliminate from program and have not any affection to original program behavior.

For example, a control flow graph shown in Figure 2.1, the first state, the basic block B_a consists of some statements or the function definitions, it has not any function-call statements or branch statements point the entry of basic block B_a. The second state, a variable i is not possibly greater than 100 after executing B_c. If it want to run into the B_b, a variable i must be greater than 100. So that, the basic block B_a and B_b never execute, surely it has not any affection to the control flow and the variables at run time, and the whole finally

results after executing. The basic block B_a and B_b are unreachable code for the control flow graph in Figure 2.1.

Unreachable code elimination analysis performs the *straightforward depth-first traversal* [DEMS00] on the control flow graph. First, all basic blocks are assigned a *unreachable-mark*, except the entry block of main program. Next, the optimization analysis traverses all interprocedural control flow graphs and verify the reachable blocks. If a block can reach from another basic blocks through the function-call, branch-jump, and flow-control, etc, it assigns a reachable-mark. It can identify unreachable block and reachable block after traversing all basic blocks and removing the unreachable blocks from control flow graph.

2.1.2 Strength Reduction

Strength reduction replaces the expensive operation with the cheap operations [Muc97]. For example, it replaces multiplication and division by additions and subtractions, it can reduce the object code size. Allen and Cocke [AllC81] discuss several applied to strength reduction as follows:

1. replacing exponentiation by multiplication,
2. division and modulo by subtractions, and
3. continuous differentiable functions by quadratic interpolations.

The follow example is a typical case for strength reduction: The sequence: 0, 3, 6, 9, 12....

It can write as S_i = 3 * i for i = 0, 1, 2, 3... . We can also replace with S_i + 1 = S_i + 3 with S_0 = 0. The second case is : The sequence:

0, 1, 4, 9, 16, 25, 36....

It can write as $S_i = i^2$ for i = 0, 1, 2, 3,.... . We can also replace with S_i + 1 = S_i + 2 * I + 1 for S_0 = 0, or as S_i + 1 = S_i + t_i where t_i + 1 = t_i + 2 , S_0 = 0 and t_0 = 1.

The cheap operations are not usually shorter than expensive operations. In some cases, the code size of the expensive operations is smaller than cheap operations. So that, the advantage of strength reduction happen on the replace operations shorter than original operations, and it can reduce the code size as soon as saving the execution time.

2.1.3 Dead Code Elimination

If the result of a variable never use on any other instructions in a program, we can say it is "dead". For example, in Figure 2.2(a), the variable j, appeared on two basic blocks, but the result of j on each exit point of basic block has not affection for program result and has not any affection for the behavior of program, whether the variable j exists in program or not. So that, the instructions relate to variable j can eliminate from the programs. The result is showed as Figure 2.2(b) after running Dead code elimination.

The instruction about variable j can eliminate on control flow graph, and reduce the object code size. Similar to the unreachable code elimination, dead code elimination also need traverse the control flow graph in order to find

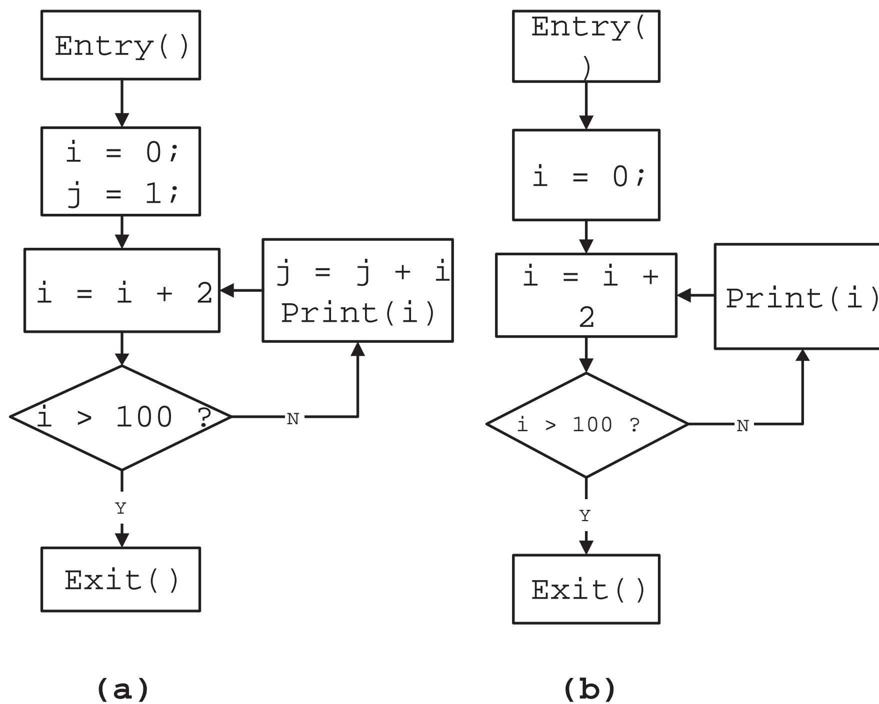


Figure 2.2: The example of Dead code elimination

all variable used states in the program and create a Mark table [Muc97] to identify the dead variables. According the table, It removes the instructions for dead variables in each basic block.

2.1.4 Redundant Code Elimination

If a computation of the expression is known and its result is computed previously and the result is confirm and fix in this time. We can say the expression instruction is redundant at that program point. If the expression of this point can be identified, we can eliminate it and have not any affecting the behavior of the program.

In Figure 2.3(a), the expression $j + 5$ appeared two times at two different

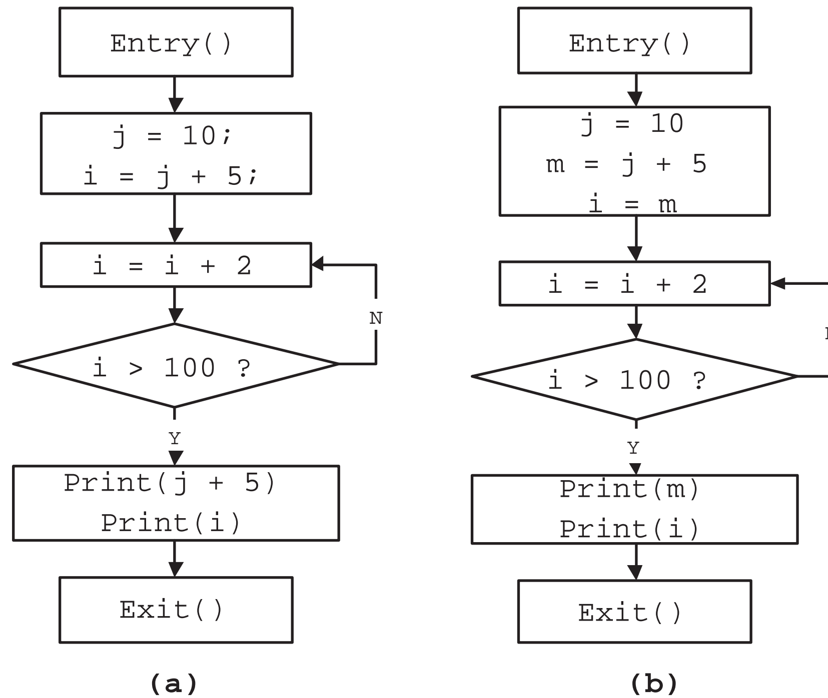


Figure 2.3: The example of Redundant code elimination

basic blocks, and result of second expression is same as first. So that, the second expression is redundant related to first expression, and it can be eliminate. Similar to previous method, we construct a table to store each computation of expressions that we want to eliminate it. In the redundant points of the program, we replace the original redundant code point with corresponding table entries, as shown in Figure 2.3(b).

2.2 Procedural Abstraction

We discuss some of the optimization techniques in section 2. The procedural abstraction is also a technique of the optimizations. We will introduce the concept for the procedural abstraction as follows :

2.2.1 Code Factoring

Code Factoring is a optimization in order to reduce the object code size. As the traditional optimizations, code factoring also reduce redundant code in order to save the code size. The idea of code factoring transformation included three main steps [DEM99] : (1) To *find* the *repeated* or *identical* code fragment from a sequence code fragments. (2) To *build* a copy of the identical code fragment so as to replace all occurring places. (3) To *arrange* for all occurring places. Some of occurring places need to insert flow transfer statement (jmp, breanch for example), and some of occurring places need to remove the unnecessary codes.

Given a example shown in Figure 2.4. We try to factor the identical code fragments using the method by moving the pre- or post-dominates all the occurring places. The left hand side of the basic block of a code fragment as the Figure 2.4(a). The block a B_a has a instruction $MOVr0, \#4$, and the block c B_c also has a instruction same as the B_a. We can see that the instruction $MOVr0, \#4$ of B_c is independent of other instructions prior to this instruction in B_c, so we can factor it out B_c into another block where place it before B_c. The instruction $MOVr0, \#4$ at both B_a and B_d, and therefore merge two block into a single block as block d B_d, shown as right hand side of Figure 2.4(b). The next case, The B_b and B_c both have the instruction $PUSHr0$. Although after of a instruction $PUSHr0$ in block has another instructions, instruction $PUSHr0$ is not predominated by following instructions. The $PUSHr0$ instructions in B_b can merge with $PUSH$ instruction in B_c into a new B_g. The

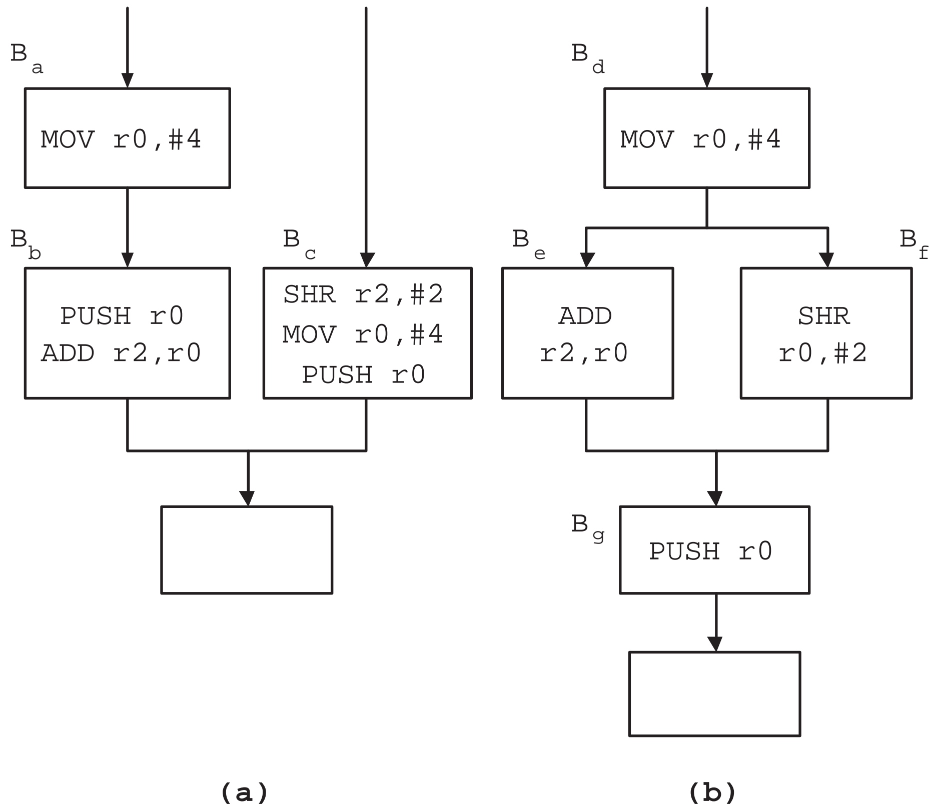


Figure 2.4: The example of Code-Factoring

result after code factoring transformation is that reducing the code size from 6-instructions to 4-instructions.

2.2.2 Cross-Jumping

Same as the procedural abstraction, the cross-jumping is a approach sharing the codes in order to reduce the code size. Cross-jumping [EEFLP97, FMW84] reuses the common tail of two merging code sequences [DEMS00]. Let see the Figure 2.5, it is a example of cross-jumping:

In Figure 2.5, there are two code fragments are identical. In this case, we need not use the procedural abstraction to transform the identical code

```
MOV r0,I              jmp sub
ADR r1,[r0]           ..
ADD r1,#10            ..
  ..                  ..
  ..                  ..
MOV r0,I        sub:  MOV r0,I
ADR r1,[r0]           ADR r1,[r0]
ADD r1,#10            ADD r1,#10
RTN                   RTN
   (a)                   (b)
```

Figure 2.5: The example of cross-jumping

fragments into a procedure and make a replacement for each original identical point. We only hold one of these identical code fragments such as second code fragment in Figure 2.5(a), and replace another identical codes with a branch statement. In Figure 2.5(b), the first code fragment transforms a JMP instruction expressing the control flow branched to label *sub*. In general, the program retains the lastly identical code fragment and transforms the other parts in order to reduce the complexity of transformation.

Figure 2.6 shows that the cross-jumping apply to control flow graph. If the basic block B_a and B_c are identical, the cross-jumping retains the B_a and transforms the B_c to B_a.

2.2.3 Procedural Abstraction

The procedural abstraction is a commonly method of code optimization. A repeated code fragment is a *single-entry* and *single-exit* code [DEMS00, FMW84], there are two steps : The first step, a compiler abstracts the code fragments and transforms into a single procedure. The second step is that replacing the

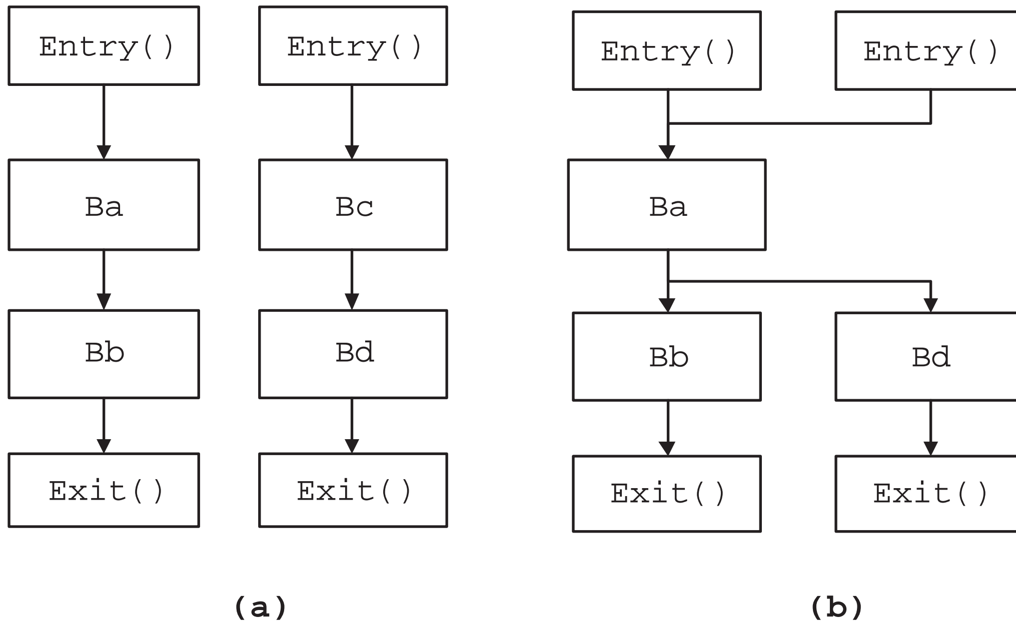


Figure 2.6: The example of Cross-jumping applied to control flow graph

original code fragments to contain the repeated code fragments with a call-function statement and to point the created procedure. The first step is a more difficult work, because of to identify the repeated code fragments is not easy to reach especially high-level stage even though there are several methods can approach it.

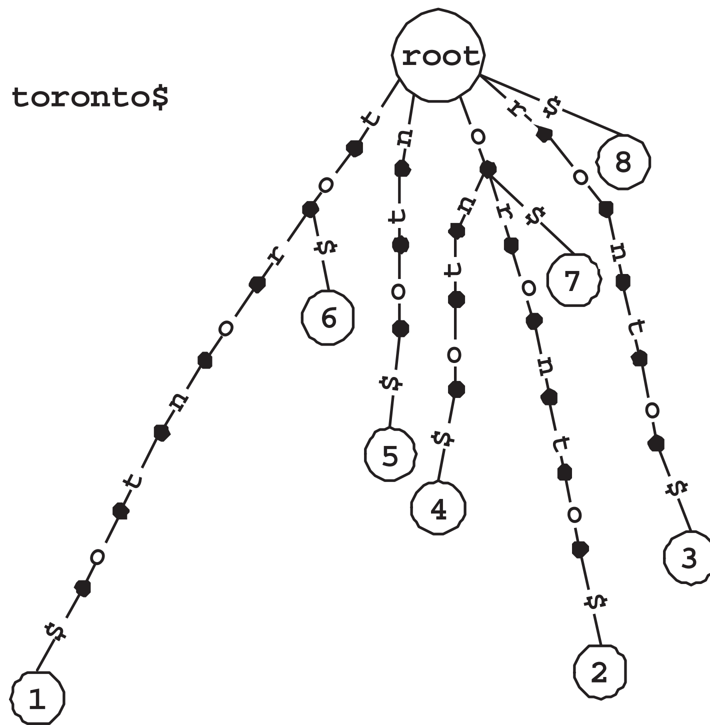
2.2.4 Identify the Repeated Code

Several methods discuss code-factoring applying to reduce the object code size. One of those methods uses the *fingerprinting* method [DEMS00] to identify whether two code fragments are identical or similar. We use the *fingerprinting function* to compute the fingerprinting word of code sequences for each basic block. The fingerprinting word is a 64-bit word that is compose of 4-bit encodings instruction opcode of the first 16 instructions in a basic block. The

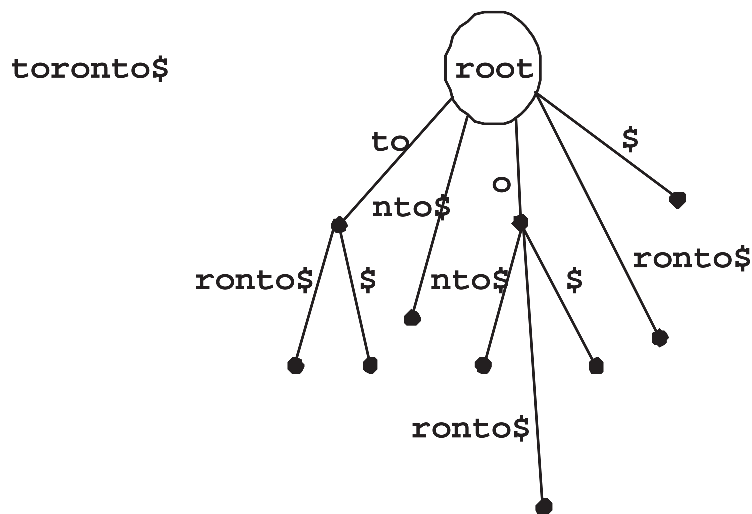
each 4-bit encoding expresses 15 varieties of different opcodes, and can present top 15 frequency instructions. If we want compare the different between the two code fragments, we can compare the two fingerprinting words of the target basic block. Hence, According the similar degree of two fingerprinting words, we can identify whether two basic blocks are equivalent or not. If the two code fragments are equal, the two code sequences are *same*, and we can use the code compression method to process them.

There are some methods to identify the identical code fragments. The *repeated table* [CM99] is a data structure to store the information about the each instruction from code sequences. For identify the repeated codes, it builds the *suffix tree* [FMW84]. The suffix tree is a kind of data structure to encode information about reduplication within textual string [CM99]. Figure 2.7 shows a detail example to construct a suffix tree for the text string *toronto*. Each inner nodes in the tree, other than the root node, identifies a repeated substring in the input, these nodes represent chances for compression, the low-level language such as assembly especially. Because of the assembly is always led off the instruction with a OP-Code and following the one to three parameters [CM99]. So that, it can get a better effective to identify the repeated strings.

In the repeated table, each entry is constructed using the suffix-tree encoding method[EEFLP97, FMW84, CM99] and includes the information that is a set of specific identical substrings within the program text. After a set information of code fragments collects into a table, the compiler reads and analyzes the table data in order to compute the identical measure of the codes.



(a)To build a Suffix TRIE for all n suffixes of the text before constructing the suffix tree.



(b)Suffix tree for the string toronto.

Figure 2.7: The example of suffix tree.

Chapter 3

Procedural Abstraction at Low-Level Stage

Procedural abstraction is an optimization technique used in many compilers. The concept of procedural abstraction derives from code-factoring, cross-jumping, and other classical code optimizations [DEMS00, FMW84, Muc97]. For procedural abstraction, we analyze on the *identical* code sequences that repeatedly occur in a program. For example, in Figure 3.1 (a), the program in the Thumb instruction set has two identical or equivalent code sequences. At the low-level stage, procedural abstraction is a familiar method for code compression [DEMS00, DEM99]. Applying the procedural abstraction transformation on the program in Figure 3.1 (a), we get a compressed program in Figure 3.1(b).

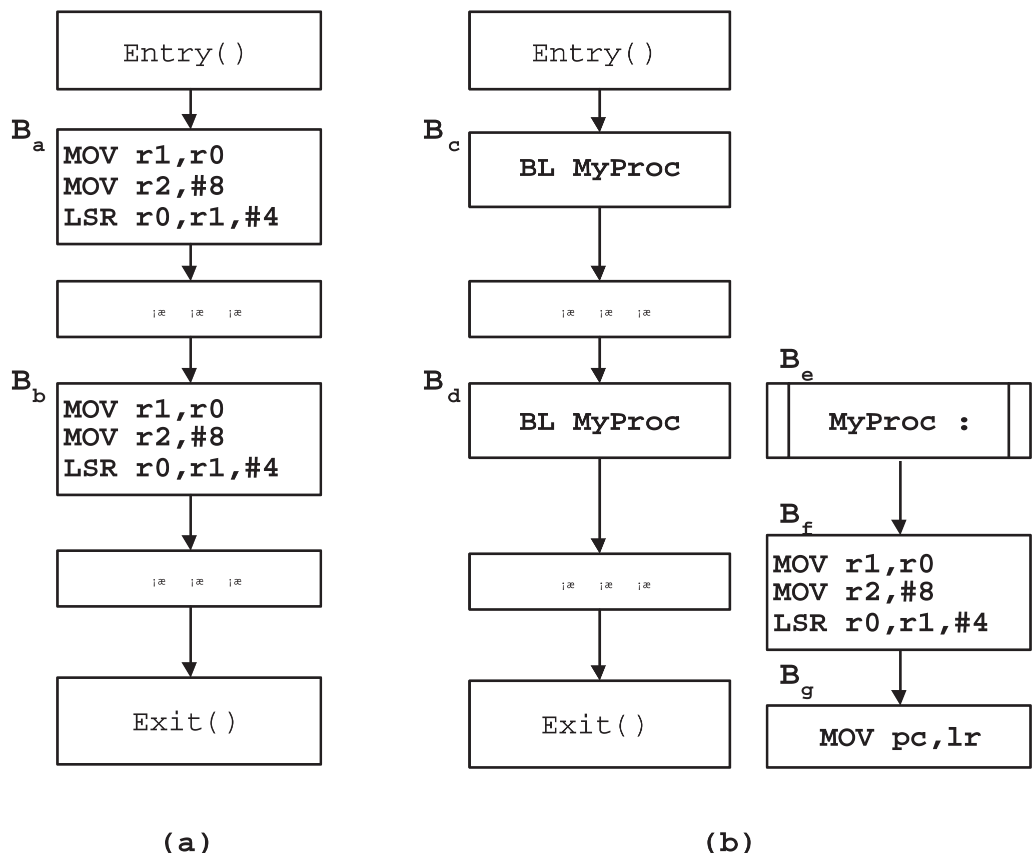


Figure 3.1: A code fragment: (a) the inline style and (b) the procedural abstracted style

3.1 Procedural Abstraction Criteria at the Low-Level Stage

Similar to the procedural abstraction at the low level stage, the compiler can factor the repeated code fragments of a high-level program into a single procedure, and replace the original occurrences of the code fragment into calls to the procedure. Hence, we reduce the code size, compared with the original object code. This technique of code compression can be implemented in the compiler optimization [FMW84].

The first steps in the procedural abstraction transformation is finding out the identical or similar code fragments such as the code fragment B_a and B_b in Figure 3.1 (a), called a *candidate code fragment*, through scanning the source program. There are a few approaches proposed for identifying the identical code fragments, such as the *fingerprinting function* [DEMS00], and the *global repeat table* [CM99] , as discussed at chapter 2.

The fingerprinting function creates a data structure for each code fragment. Each instruction in the code fragment assignes a binary value, called *fingerprint*, according its opcode and operands. If the fingerprints of two code fragments are the same or similar, these code fragments identify as *identical* or *equivalent*, respectively [DEM99].

After identifying the candidate code fragments, the compiler factors out the code fragment into a single procedure. For example, it factors the identical code fragment into B_e, B_f, and B_g in Figure 3.1 (b), and uses the corresponding procedure calls B_c and B_d in Figure 3.1 (b) in the place of the original code fragments. The procedure calls introduce some overhead by adding the procedure header and return instructions. On the other hand, the size of the object code reduces if code size of the procedure calls is smaller than the size of the identical code fragments. For example, the B_a and B_b code size on procedural mode in Figure 3.1 (a) is 6 bytes equal to the code size on inline mode in Figure 3.1 (b) from B_c to B_g. Because the repeated time of some cases are too small to result in the overhead exceeds the saving spaces in procedural abstraction mode, we know there are some cases is that the object code size

in procedural abstraction mode is smaller than the object code size in inline mode. If the repeated time is greater than 2, the procedural abstraction reduces the object code size. In this chapter, we will find the suitable repeated time to know the procedural abstraction can reduce the object code size.

In general, procedural abstraction can reduce the size of object codes in a lot of cases. But, there are some cases that procedural abstraction is not able to reduce the code size. For the purpose of code compression at the low-level stage, we experiment on the effect of the procedure abstraction. We propose the *procedural abstraction model* for deciding whether the procedural abstraction reduces code size. We also present the criteria, called *procedural abstraction criteria*, such that if the criteria satisfies procedure abstraction transformation will reduce the code size.

The following section presents our experiments on the effect of procedural abstraction at the low-level stage. We propose the procedural abstraction criteria in section 3.3. Section 3.4 describes the procedural abstraction model.

3.2 Experiments

We experiment on the effect of procedural abstraction on the assembly code in the ARM/Thumb instruction set. The experiment includes seven Thumb assembly programs. Each program is implemented in two styles: the *inline style* and the *procedural abstracted style*. We use the *Thumb instruction set* of ARM assembly to implement our experiment and use the ARM/Thumb development tool to develop and simulate the program [ARM98b, ARM98a].

The programs are compiled and linked using the ARM Software Development Toolkit 2.51 [ARMh].

The ARM Software Development Toolkit 2.51 is an integrated development environment to develop the ARM processor software program [ARM98a, ARMh]. The main components include follows:

1. Command-line development tools.
2. Windows development tools.
3. Utilities.
4. Supporting software.

Our experiment builds on ARM Project Manager in the ARM SDT 2.51. It includes assembly editor, C editor,*armcc* (The ARM C compiler),*tcc* (The Thumb C compiler),*armasm* (The ARM and Thumb assembler),*armlink* (The ARM linker), and *armsd* (The ARM and Thumb symbolic debugger). It is a graphic user interface tool that automates the routine operations of managing source files and building software development projects [ARM98a]. This experiment uses the ARM Project Manager and the ARM Debugger for Windows in ARM ADT2.51 shown in Figure 3.2 and Figure 3.3. The details of the experiment platform are listed as follows:

1. OS : Microsoft Windows 2000
2. Development Tools : ARM Software Development Toolkit 2.51
   (a) The ARM Project Manager.

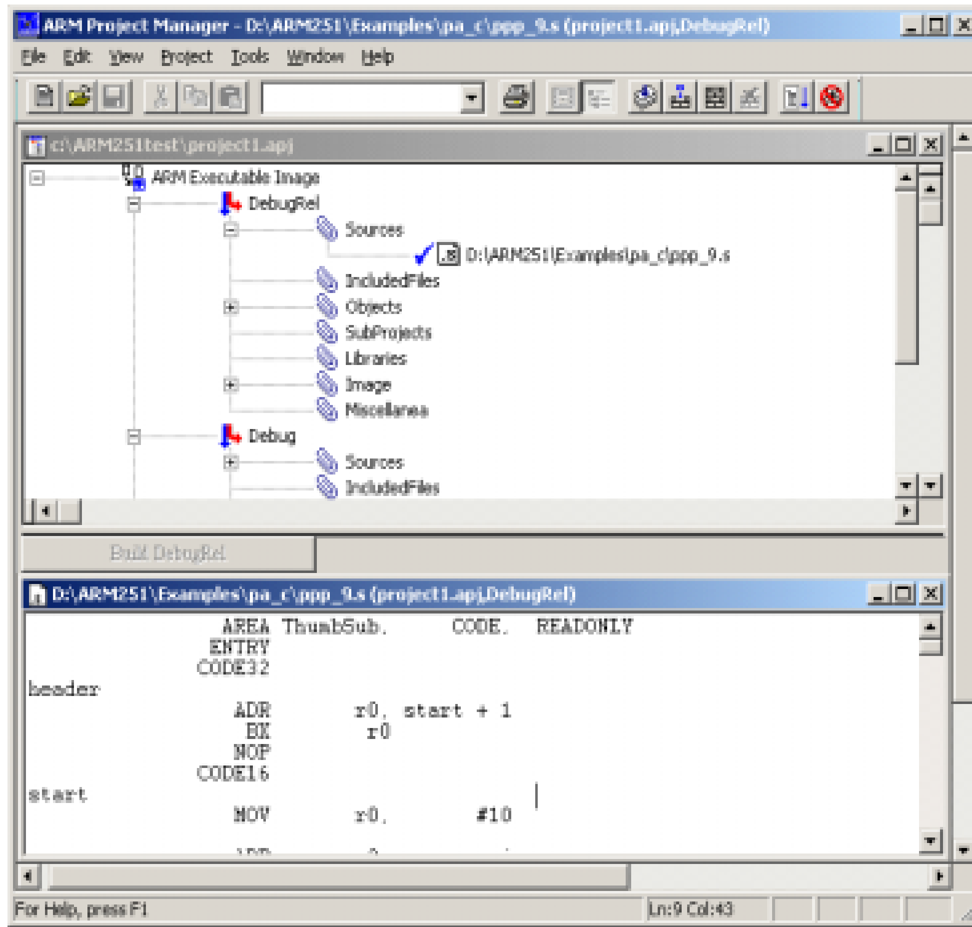


Figure 3.2: The ARM Project Manager

(b) The ARM Debugger for Windows.

We calculate the sizes, in bytes, of the object code for all programs. For each program, we experiment on the effect of procedural abstraction with the candidate code fragment appeared in the program from one to seven times. The benchmarks are collected in Figure 3.4. In the figure, the first row is the number of lines in the candidate code fragment.

Note that the code sizes for the procedural abstracted programs with a one-line candidate code fragment is the same as that with a two-line candidate

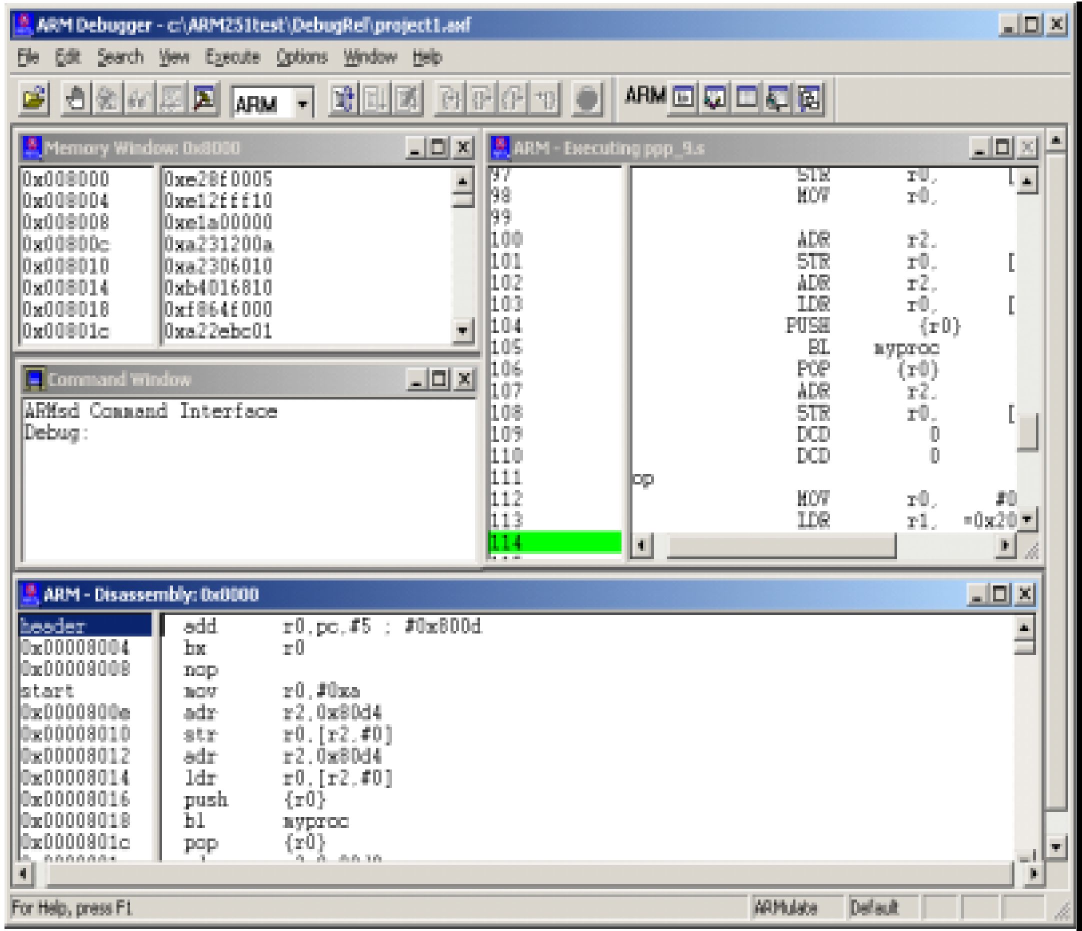


Figure 3.3: The ARM Debugger for Windows

# of lines	1		2		3		4		5		6		7	
# of occurrences	INL	PA	INL	PA	INL	PA	INL	PA	INL	PA	INL	PA	INL	PA
1	28	36	32	36	32	40	36	40	36	44	40	44	40	48
2	36	44	40	44	44	48	48	48	52	52	56	52	60	56
3	40	52	48	52	52	56	60	56	64	60	72	60	76	64
4	48	60	56	60	64	64	76	64	80	68	88	68	96	72
5	52	68	64	68	72	72	88	72	92	76	104	76	112	80
6	60	76	72	76	84	80	100	80	108	84	120	84	132	88
7	64	84	80	84	92	88	112	88	120	92	136	92	148	96

Figure 3.4: Benchmarks from the experiments on the procedural abstraction criteria at the low level stage

code fragment in Figure 3.4. The reason is that when *armlink*, the linker of ARM Software Development Kit [ARMh], builds the execution code from the assembly code, it automatically adjusts the object code into the 4-byte words by adding useless instructions, such as *lsl r0,#0*, into the original assembly codes.

3.3 Procedural Abstraction Criteria

Summarizing from the benchmarks in Figure 3.4, we derive a formula for the code size of the in-line program with a one-line candidate code fragment.

$$\mathcal{C}_i = \lceil \frac{22 + 6n}{4} \rceil \times 4,$$

where $\mathcal{C}_i$ is the code size of the in-line program, and n is the number of occurrences of the candidate code fragments. For examples:

$\mathcal{C}_i = \lceil \frac{22+6}{4} \rceil \times 4 = 28$ when $n = 1$, and

$\mathcal{C}_i = \lceil \frac{22+6\times 2}{4} \rceil \times 4 = 36$ when $n = 2$.

For the procedural abstracted program with a one-line candidate code fragment in Figure 3.4, the derived formula for the code size is as follows.

$$\mathcal{C}_p = 28 + 8n,$$

where $\mathcal{C}_p$ is the code size of the procedural abstracted program, and n is the number of occurrences of the candidate code fragments. For examples:

$\mathcal{C}_p = 28 + 8 = 36$ when $n = 1$, and

$\mathcal{C}_p = 28 + 8 \times 2 = 44$ when $n = 2$.

Based on the benchmarks from the experiments, we derive the procedural abstraction criteria, which is presented as an algorithm shown in Figure 3.5. We may easily implement the algorithm as a function in the compiler optimization module to detect where procedural abstraction may reduce the code size. The algorithm has two input parameters:

lines is the number of lines in the candidate code fragment; and *times* is the number of occurrences of the candidate code fragment in the program.

If the algorithm returns *true*, then procedural abstraction transformation will reduce the code size. Otherwise, the compiler keeps the program in the in-line style.

Our experiments are based on the Thumb instruction set. Such experiments may be easily adapted to the systems using other instruction sets.

3.4 Procedural Abstraction Model

Extending from the procedural abstraction criteria at the low level stage, we propose a model, *procedural abstraction model*, to describe the behavior of the size change with respect to the number of the occurrences for the candidate code fragment.

Suppose a candidate code fragment appears n times in a program. Let $\mathcal{S}_1$ be the object code size for the candidate code fragment, $\mathcal{S}_2$ be the size overhead [ASU86] for making the candidate code fragment a procedure, and $\mathcal{S}_3$ be the size for a call to the procedure. The n occurrences of the candidate code fragment take the size of $n \times \mathcal{S}_1$ in the object code of the in-line program,

```
Procedural_Abstraction_Criteria
    'Description :
          'Applying to Low-Level(Assembly Level) stage
          'Read times and lines as input
          'in order to evaluate whether PA mode
          'can be reduced the code size
    'Input :
          'times: Repeat times of repeated code fragment
          'lines: lines (Size) of instruction
                'Each instruction size equal to 2 bytes
    'Output:
          TRUE : for reducing code size at PA mode
  {
   select (lines)
    when 3:
     if times ≥ 6
           return {true};
    when 4:
     if times ≥ 3
           return {true};
    when > 4:
     if times ≥ 2
           return {true};
    otherwise :
     return {FALSE};
   end select
  }
```

Figure 3.5: The algorithm for the procedural abstraction criteria at the low level stage

while they take the size of $(\mathcal{S}_1+\mathcal{S}_2)+n\times\mathcal{S}_3$ in the object code of the transformed program after procedural abstraction.

Hence, given a program in which a candidate code fragment appears n times, we may compute if there exists a k such that

$$\begin{aligned} k \times \mathcal{S}_1 &\leq (\mathcal{S}_1 + \mathcal{S}_3) + (k \times \mathcal{S}_2), \text{ and} \\ (k+1) \times \mathcal{S}_1 &> (\mathcal{S}_1 + \mathcal{S}_3) + (k+1) \times \mathcal{S}_2. \end{aligned}$$

If so, k is the procedural abstraction criteria for the candidate code fragment in the program.

To summarize, we present the problem of finding the procedural abstraction criteria as follows.

> **Definition (procedural abstraction criteria problem)**:Given a program in which a candidate code fragment appears n times, compute if there exists a k such that
>
> $$\begin{aligned} k \times \mathcal{S}_1 &\leq (\mathcal{S}_1 + \mathcal{S}_3) + (k \times \mathcal{S}_2), \text{ and} \\ (k+1) \times \mathcal{S}_1 &> (\mathcal{S}_1 + \mathcal{S}_3) + (k+1) \times \mathcal{S}_2, \end{aligned}$$
>
> where $\mathcal{S}_1$ is the size for the candidate code fragment, $\mathcal{S}_2$ is the size overhead for making the candidate code fragment a procedure, and $\mathcal{S}_3$ is the size for a call to the procedure. □

Note that if such a k exists, procedural abstraction actually reduces the code size for the program in which $n > k$.

Chapter 4

Procedural Abstraction Criteria at the High-Level Stage

Procedural abstraction is a common code compaction technique of the compiler optimizations[DEMS00]. In previous chapter, we apply the procedural abstraction to the code compression at the low-level stage such as assembly language. We propose a model for deciding whether the procedural abstraction transformation on the source program may reduce the size of object code. And, the criteria for the low-level stage is proposed for the procedural abstraction to reduce object code size. We also include the benchmarks of the experiments on our approach of the procedural abstraction at the low-level stage.

In general, these code optimization is usually used in the low level that the instruction formats are fix, regular, and easy to analyze. In this chapter, we extend the research areas to the high-level stage. The same as low-level

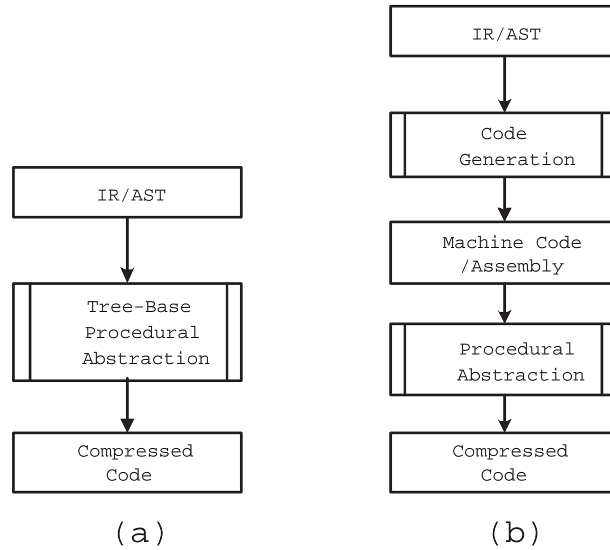


Figure 4.1: The structure diagram of applying procedural abstraction: (a) tree-based approach (b) conventional approach

procedural abstraction criteria, we propose a model for determining whether the procedural abstraction transformation on the source program (high-level statement like C, Java) may reduce the size of object code (low-level code).

We implement our approach in a simple *C-to-Thumb* compiler. In the previous works, the procedural abstraction usually implements at the low-level stage, such as the assembly level [DEMS00, Luc00, BZIP2h]. The overall process is shown in Figure 4.1 (b). In contrast to those approaches, we apply procedural abstraction to the high-level stage. The procedural abstraction transformation performs on the abstract syntax trees of the source programs. We call the technique a *tree-base procedural abstraction*, as shown in Figure 4.1 (a).

```
main()                  int myproc(int j)
{                       {
   int i;                 j = j + 5;
   int s;                 return j;
   i = 0;               }
   s = i + 5;           main()
   .                    {
   .                      int i;
   s = i + 5;             int s;
   .                      i = 0;
}                         s = myproc(i);
                          .
                          .
                          s = myproc(i);
                          .
                        }
(a)                       (b)
```

Figure 4.2: A simple program in C (a) the inline style (b) the procedural abstracted style

4.1 Procedural Abstraction Criteria at the High-Level Stage

We apply the procedural abstraction technique to the high-level stage. That is, we use the procedural abstraction criteria to decide whether the procedural abstraction reduces the code size for a program in the high-level programming languages, such as C, C++, and Java. Similar to the procedural abstraction at the low-level stage, the high-level procedural abstraction factors out the identical code fragment into a single procedure, and replaces the code fragments with calls to the procedure.

As an example, we consider a simple programs in C shown in Figure 4.2. Figure 4.2 (a) is a program in the inline style. It appears that the C statement $s = i + 5$ occurs two times. We can say that the statement $s = i + 5$

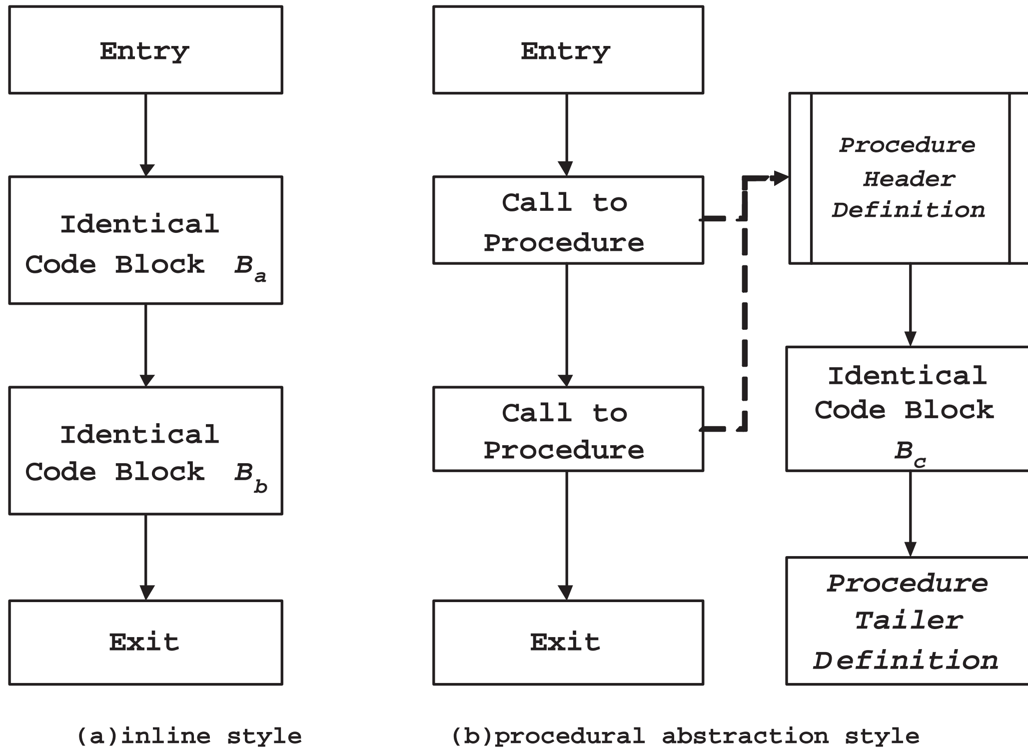


Figure 4.3: The examples of the inline style and the procedural abstracted style

is an identical statement [CM99], which is also called as a repeated statement [DEM99], or an equivalent statement [DEMS00]. If we can abstract the repeated statement into a procedure and the original statement are replaced the *caller statement* such as *function call statement* direct to the procedure, we can reduce the several code statements that the effect of space saving somewhere is a multiple of the size of the repeated statement. About the example in Figure 4.2, if the operation appears several times, it is merited to turn the program into the procedural abstracted style, as shown in Figure 4.2 (b).

The repeated statement is sometime not only a single statement or express, but it is also a several of statements. In Figure 4.3, we view the repeated statements as a *basic block* in the control flow graph. Such as the B_a and B_b

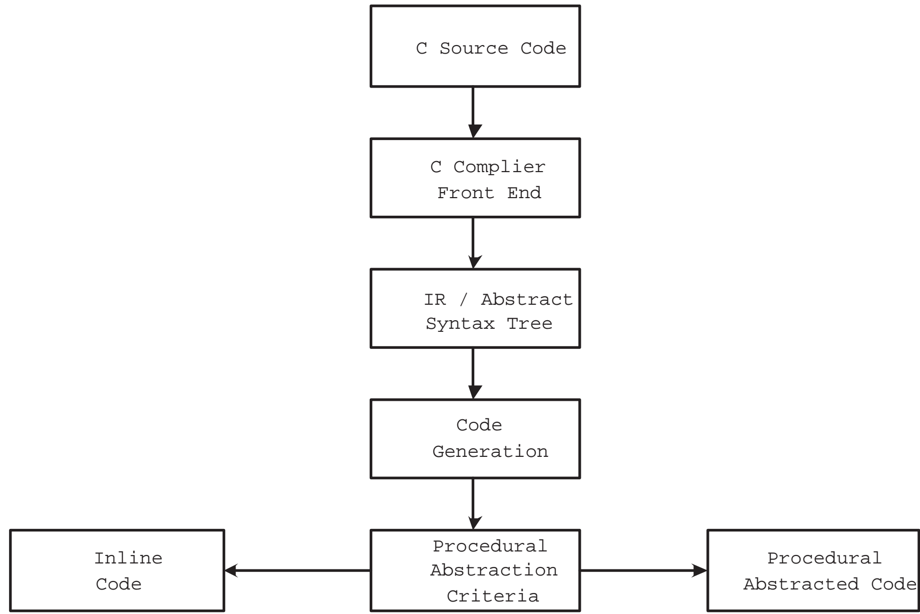


Figure 4.4: The structure diagram of applying procedural abstraction at the high level stage

in Figure 4.3 (a). Similar to Figure 4.2, if B_a and B_b are not exactly identical, but they are mostly similar, we can say that the basic blocks B_a and B_b are equivalent code fragments. To make use of the concept of the procedural abstraction at Figure 4.3 (a), it abstracts the blocks B_a and B_b into a procedure block B_c. There is some overhead in order to produce a procedure, such as adding procedure headers and tailers. The identical block in the original program is replaced by the function call statement to the abstracted procedure. The final result is showed in Figure 4.3 (b).

We investigate on finding the procedural abstraction criteria at the high-level stage. Due to the fact that it is not easy to estimate the size of the object codes for each high level statement, we apply a tree-based approach for finding the procedural abstraction criteria at the high-level stage, called

tree-based procedural abstraction criteria.

The structure of applying the tree-based procedural abstraction is presented in Figure 4.4 . Instead of taking the source code of a program as input, the tree-based procedural abstraction criteria takes an *abstract syntax tree* (AST) as input, and computes the procedural abstraction criteria based on the size of the object code generated from the AST.

4.2 The C-to-Thumb Compiler

In this thesis, we construct an experimental C-to-Thumb compiler in order to perform our experiments. The notions and techniques for constructing our C-to-Thumb compiler refers to several famous and learned compilers such as follows :

1. *GNU gcc Compiler*, and other GNU development software packages of programming language and compiler [LO95, GNUh, GCCh, Pra02],
2. *EAS Compiler* [Yun98],
3. *Tiger Compiler* [AG97],
4. *lcc Compiler (with BURG Code Generation)* [FH95, FHP92], and
5. *Small C Compiler* [Hen90].

4.2.1 Compiler Framework

In our compiler implementation and experiments, we use the ANSI C specification and grammar from Jeff Lee [Lee85] as the lex specification and the

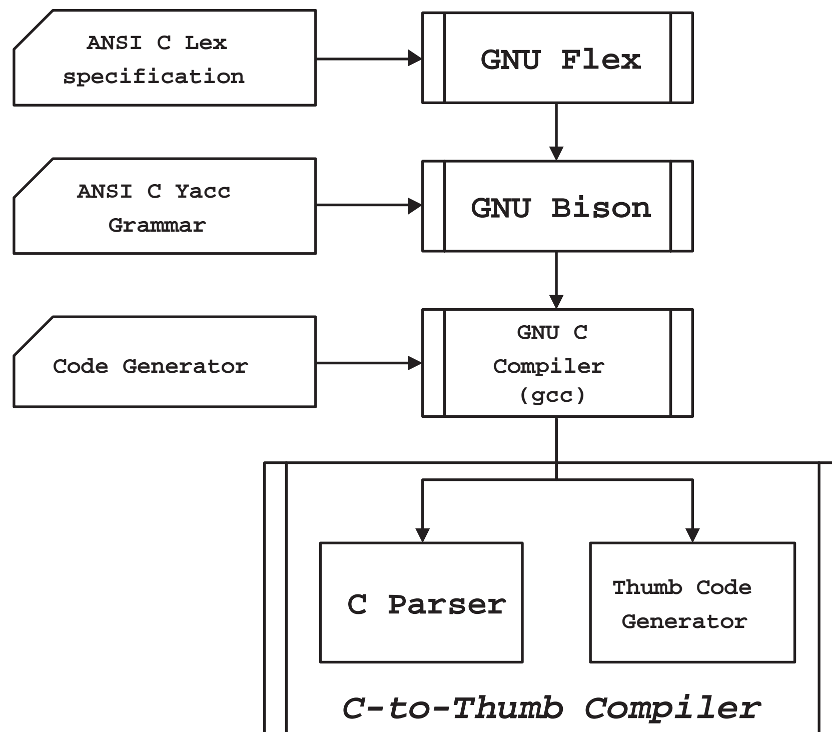


Figure 4.5: The process to construct a C-to-Thumb Compiler.

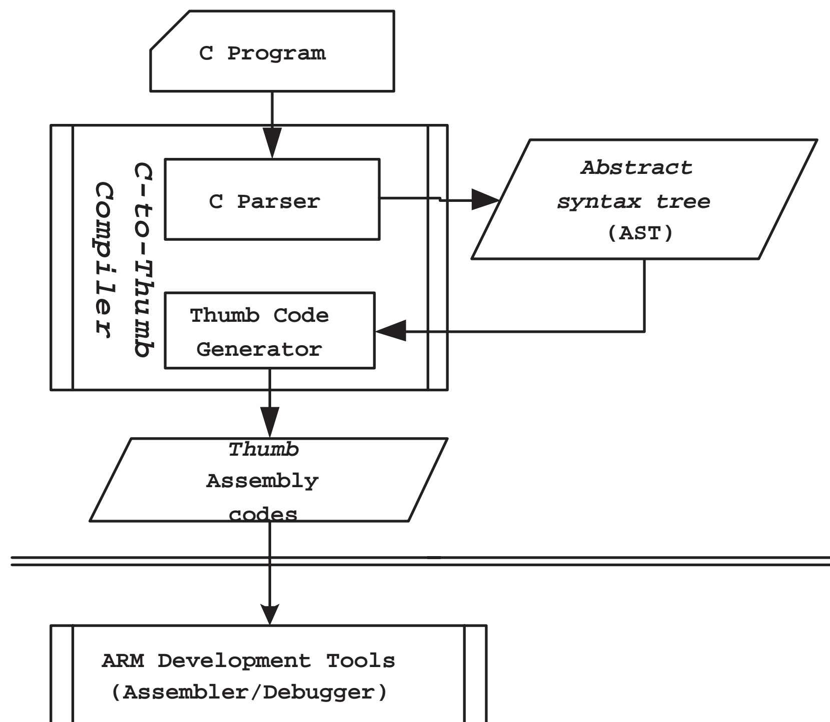


Figure 4.6: Using the C-to-Thumb Compiler to compile the C source program.

yacc grammar of our compiler. The structures of C-to-Thumb compiler are shown in Figure 4.5. The compiler uses the development tools including the *GUN-Flex, GNU-Bison and GNU gcc compiler* [LMB92]. According to the compiler requirements, we write the *semantic action* programs into the yacc grammar file. We get the executable compiler after compiling by gcc compiler. The compiler contains two components : *C parser* and *Thumb code generator*, shown as Figure 4.5 :

1. C parser : The C parser reads a C source program as input, and parses it into an *abstract syntax tree (AST).*

2. Thumb code generator : The AST is passed into the code generator. The code generator generates the Thumb assembly codes.

The operation of the C-to-Thumb compiler is shown in Figure 4.6. It compiles the C source codes and generates the Thumb assembly codes directly. In order for the compiler to be easily implemented, we omit some optimizations, including some of the error checking, and some of the control flow analysis.

In Figure 4.6, the compiler reads an ANSI C program as input, and the parser generates the abstract syntax trees. This stage also generates the *symbol tables* which are stored the data for the variables, the declarations of types, and the function definitions. The second stage of the compiler is a code generator. The ASTs are generated from parsing stage, and they are passed into the code generator. The code generator generates the Thumb assembly codes directly. The C-to-Thumb is a simple and direct compiler, compared with the other mature and full-function compilers.

4.2.2 Compiler Developmental Environment

The compiler is developed based on the ANSI C grammar written in the *lex/yacc* specification and compiled by GNU flex/bison [LMB92]. The compiler reads C source codes, and constructs the abstract syntax trees, and generates the ARM/Thumb assembly codes. The development environment of our experimental compiler is listed as follows :

1. OS platform : Mandrake Linux 8.2.
2. Development Tools :
    (a) GNU C/C++ 2.96 compiler.
    (b) GNU flex/bison.

3. Lex/Yacc specification : ANSI C grammar and specification [Lee85].

4.2.3 Compiler Specification

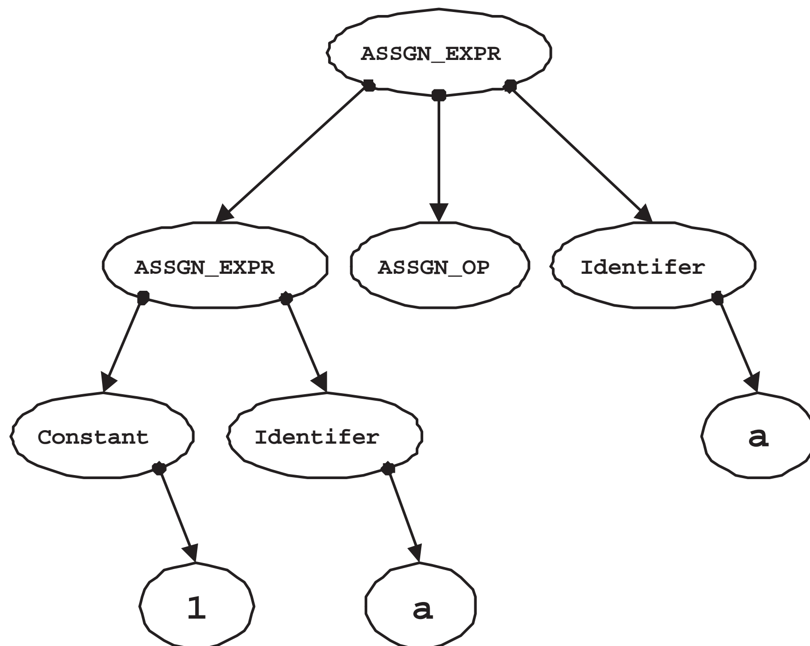
Without loss of generality, we only implement part of ANSI C in our C-to-Thumb compiler. The specification for the compiler is listed as follows:

1. Data type :
   (a) integer.
   (b) character.
2. Function :
   (a) Function Definition(with the parameters, and without the parameters)
   (b) Function call.
3. Control flow statements :
   (a) while-loop statement.
   (b) do-loop statement.
   (c) for-loop statement.
   (d) if-then-else statement.
   (e) switch statement.
   (f) goto statement.
4. Operators : arithmetic operators, and logical operators.
5. Expression statements.

Figure 4.7: Using the C-to-Thumb Compiler (Snapshot 1).

```
****Global Symbol Table & Code Printing ...(start)
                AREA ThumbSub,      CODE,  READONLY@( 0)@
                ENTRY                                @( 0)@
                CODE32                               @( 0)@
header                                               @( 0)@
                ADR       r0, start + 1              @( 0)@MainProg-Begin
                BX        r0                         @( 2)@
                NOP                                  @( 2)@
                CODE16                               @( 0)@
start                                                @( 0)@
                ADR       r2,        a               @( 0)@Express-statement
                LDR       r0,      [r2]              @( 2)@
                MOV       r1,        #1              @( 2)@
                                                     @( 0)@
                ADD       r0,       r0,         r1@( 2)@
                ADR       r2,        a               @( 0)@
                STR       r0,      [r2]              @( 2)@
a               DCD        0                         @( 0)@
stop                                                 @( 0)@
                MOV       r0,     #0x18              @( 2)@MainProg-End
                LDR       r1,  =0x20026              @( 2)@
                SWI      0xAB                        @( 2)@
                MOV       pc,        lr              @( 2)@Function-end
                END                                  @( 0)@
****Global Symbol Table & Code Printing ...(end)
****Final Code Size equal to ..............20
[eeyore@eeyore ansi]$
```

Figure 4.8: Using the C-to-Thumb Compiler (Snapshot 2).



(a)A parser-tree of the statement a = a + 1

```
    .
    .
ADR r2,a
LDR r0,[r2]
MOV r1,#1

ADD r0,r0,r1
ADR r2,a
STR r0,[r2]
    .
    .
```

(b)The Thumb Assembly codes are generated by C-to-Thumb compiler

Figure 4.9: Using the C-to-Thumb Compiler (the output of the compiler).

4.2.4 An Example

We demonstrate out C-to-Thumb compiler with an example of compiling a C program into a Thumb program. The example consists of a simple C statement : $a = a + 1;$. Figure 4.7 and Figure 4.8 show the main processes of compiling the example program. The process of constructing the ASTs is shown in Figure 4.7, and the process of generating the Thumb assembly codes is shown as Figure 4.8.

The AST constructed by the compiler is shown in Figure 4.9, and it is the tree-based structure to express $a = a + 1;$. In this example, the compiler generates six Thumb codes for the statement, shown as Figure 4.9.

4.3 Experiments

For experiments on the procedural abstraction at the high-level stage. The structure diagram of our model is shown in Figure 4.4. Our experiment includes 5 sets of simple C programs. Each set includes the programs with the same candidate code fragment occurred from 1 to 8 times. We use the sets of programs to experiment on the effect of the code size made by the different sizes of candidate code fragments. Each program in each set is compiled two times : In the first time, it is compiled without procedural abstraction, in the second time, it is compiled with procedural abstraction. The code sizes of the programs are calculated, shown as Figure 4.10.

	Program 1		Program 2		Program 3		Program 4		Program 5	
size of candidate statement sequence	22		30		38		44		52	
# of occurrences	INL	PA	INL	PA	INL	PA	INL	PA	INL	PA
1	48	80	56	88	63	96	76	100	84	108
2	68	100	84	108	96	116	120	120	136	128
3	88	124	112	132	132	140	168	144	192	152
4	108	144	140	152	164	160	212	164	224	172
5	128	168	168	176	200	184	260	188	300	196
6	148	188	196	196	232	204	304	208	352	216
7	168	212	224	220	268	228	352	232	408	240
8	188	232	252	240	300	248	396	252	460	260

Figure 4.10: Benchmarks from the experiments on the procedural abstraction criteria at the high level stage

4.4 Procedural Abstraction Criteria

Based on the benchmarks in Figure 4.10, we get the following observations, in a conservative sense, according to the size of the candidate code fragment:

- If the size of the candidate code fragment is less than 30 bytes, the procedural abstraction transformation increases the size of the object code.
- If the size of the candidate code fragment is between 30 and 37 bytes, the procedural abstraction transformation reduces the size of the object code if the candidate code fragment appears 7 or more times.
- If the size of the candidate code fragment is between 38 and 43 bytes, the procedural abstraction transformation reduces the size of the object code if the candidate code fragment appears 4 or more times.
- If the size of the candidate code fragment is between 44 and 51 bytes,

the procedural abstraction transformation reduces the size of the object code if the candidate code fragment appears 3 or more times.

- If the size of the candidate code fragment is larger than 51 bytes, the procedural abstraction transformation reduces the size of the object code if the candidate code fragment appears 2 or more times.

We conclude the procedural abstraction criteria at the high level stage as an algorithm shown in Figure 4.11, which we implement in the procedural abstraction module of the experimental compiler.

4.5 Procedural Abstraction Criteria Model at the High-Level Stage

According to the previous discussion about the procedural abstraction at the high-level stage, we propose a model, *procedural abstraction criteria model at the high-level stage*, to describe the behavior of the generated code size change with respect to the number of the occurrences of the candidate high-level program statements sequence.

Suppose an abstract syntax tree T for a single candidate high-level language statement in a program. Let the size of the generated codes for T, $\mathcal{S}_1(T)$, is computable. Let $S2(T)$ be code size of the overhead for making T a procedure, and $S3(T)$ be the size for a call statement to the procedure replacing the original statement. The n occurrences of the candidate high-level statement take the size of $n \times \mathcal{S}_1(T)$, while they take the size of $(\mathcal{S}_1(T)+\mathcal{S}_2(T))+n\times\mathcal{S}_3(T)$

```
Procedural_Abstraction_Criteria
    'Description :
        'Applying to High-Level(Parse-Tree Level) stage.
        'Read times and Size as input
        'in order to evaluate whether PA mode
        'can be reduced the code size.
    'Input :
        'times: Repeat times of repeated statement.
        'size: Size of instruction.
            'Each instruction size equal to 2 bytes.
            'Calculated from C-to-Thumb Compiler.
    'Output:
        TRUE : for reducing code size at PA mode.
  {
   select (sizes)
    when between 30 and 37:
      if times ≥ 7
        return {true};
    when between 38 and 43:
      if times ≥ 4
        return {true};
    when ≥ 44:
      if times ≥ 2
        return {true};
    otherwise :
      return {FALSE};
   end select
  }
```

Figure 4.11: The algorithm for the procedural abstraction criteria at the high level stage

in the statement of the transformed program after procedural abstraction. The $\mathcal{S}_1(T)$, $\mathcal{S}_2(T)$ and $\mathcal{S}_3(T)$ are computed on the code generated from the abstract syntax tree T by our compiler.

Therefore, given a program in which a candidate statement occurs n times, we may compute if there exists a k such that

$$\begin{aligned} k \times \mathcal{S}_1(T) &\leq (\mathcal{S}_1(T) + \mathcal{S}_3(T)) + (k \times \mathcal{S}_2(T)), \text{ and} \\ (k+1) \times \mathcal{S}_1(T) &> (\mathcal{S}_1(T) + \mathcal{S}_3(T)) + (k+1) \times \mathcal{S}_2(T). \end{aligned}$$

If so, k is the high-level procedural abstraction criteria for the candidate statement in the program.

Now, our procedural abstraction model extends from the single high-level statement to a high-level statement sequences. Suppose a abstract syntax tree T_i for i high-level language statements ($statement_1$, $statement_2 \ldots statement_i$) composed of each individual parse-tree ($T_1 \ldots T_i$) for the individual statement. The $\mathcal{S}_1(T)$, it expresses the size of the generated assembly code for abstract syntax tree T about the singe statement, we extend to $\mathcal{S}_1(T_i)$ that expressed the sum of $\mathcal{S}_1(T_1)$, $\mathcal{S}_1(T_2)$ until $\mathcal{S}_1(T_i)$ about a high-level statement sequences, shown in follow:

$$\mathcal{S}_1(T_i) = \sum_{j=1}^{i} \mathcal{S}_1(T_j).$$

Similar to $\mathcal{S}_1$, we extend, the $\mathcal{S}_2(T)$ corresponding to $\mathcal{S}_2(T_i)$ and $\mathcal{S}_3(T)$ corresponding to $\mathcal{S}_3(T_i)$, for the cases of the high-level statement sequences.

Given a high-level program in which a candidate statement sequences occurs n times, we may compute if there exists a k such that

$$\begin{aligned} k \times \mathcal{S}_1(T_i) &\leq (\mathcal{S}_1(T_i) + \mathcal{S}_3(T_i)) + (k \times \mathcal{S}_2(T_i)), \text{ and} \\ (k+1) \times \mathcal{S}_1(T_i) &> (\mathcal{S}_1(T_i) + \mathcal{S}_3(T_i)) + (k+1) \times \mathcal{S}_2(T_i). \end{aligned}$$

If so, k is the high-level procedural abstraction criteria for the candidate statement sequences in the program.

To summarize, we present the problem of finding the procedural abstraction criteria at high-level stage as follows.

> Definition (high-level procedural abstraction criteria problem): Given a program in which a candidate statement sequence $statement_1 \ldots statement_i$ occurred n times, compute if there exists a k such that
>
> $$\begin{aligned} k \times \mathcal{S}_1(T_i) &\leq (\mathcal{S}_1(T_i) + \mathcal{S}_3(T_i)) + (k \times \mathcal{S}_2(T_i)), \text{ and} \\ (k+1) \times \mathcal{S}_1(T_i) &> (\mathcal{S}_1(T_i) + \mathcal{S}_3(T_i)) + (k+1) \times \mathcal{S}_2(T_i), \end{aligned}$$
>
> where the abstract syntax tree T_i is composed of each individual sub-tree $T_1 \ldots T_i$ for the statement sequence $statement_1 \ldots statement_i$ respectively. And, $\mathcal{S}_1(T_i)$ is the size for the candidate statement sequence, $\mathcal{S}_2(T_i)$ is the size overhead for making the candidate statement sequence a procedure, and $\mathcal{S}_3(T_i)$ is the size for a call to the procedure. □

Note that if k exists, high-level procedural abstraction actually reduces the size of the object code when $n > k$.

According the above definition for the high-level procedural abstraction criteria, we construct an algorithm, shown in Figure 4.12, to estimate whether the procedural abstraction can reduce the code size for the specific statement sequences or not. The algorithm, *Estimate_criteria*, reads the abstract syntax trees of a statement sequences (T_i) that we want estimate as input, and it returns three values. First, *candNum*, is the high-level procedural abstraction criteria for the given candidate statement sequences (T_i), which is the same as the k in above model. The second return value S_p expresses the assembly code size of the candidate statement sequences when the statement sequences to be in procedural abstraction mode, and the third return value S_i expresses the assembly code size of the candidate statement sequences when the statement sequences is kept as inline mode.

If we want to know which number of procedural abstraction criteria (k) can reduce the code size at the high-level procedural abstraction mode, we can take the abstract syntax trees of the statement sequences form compiler as input, and use the algorithm to run it. After executing, if the return value *candNum* (procedural abstraction criteria) is greater than one, it mean that reducing the code size at PA mode and the code size equals S_p when the statement sequences at original program occurring times is greater than and equal to *candNum*. If the *candNum* equals one, it cannot reduce the code size in PA mode, and we also get the code size in PA mode and inline mode individually from S_p and S_i.

In the algorithm Estimate_criteria have to call a procedure *StmSize*, shown

```
Algorithm Estimate_criteria(T_i : List of Stm-Tree)
    S_1:integer //Code Size of candidate statements
    S_2:integer //Code Size of Overhead for Factoring
    S_3:integer //Code Size of Function-Call statements
    S_i:integer //Code Size at inline mode
    S_p:integer //Code Size at PA mode
    candNum:integer //Number of PA criteria

    For Each T_j in T_i
        S_1←S_1+StmSize(T_j)
        ReadNext(T_i)
    end-For

    S_p←StmSize("ProcedureHead")+StmSize("ProcedureTail")
    S_p←S_p+S_1
    candNum ← 1

    while (candNum × S_1
            ≤ S_1 + candNum × (S_3+ S_2))
        candNum ← candNum + 1
    end-while

    S_p ← S_1 + candNum × (S_3+ S_2)
    S_i ← candNum × S_1
    return ⟨ S_p ,S_i ,candNum ⟩
end
```

Figure 4.12: The algorithm for estimate the high-level Procedural abstraction criteria.

```
TYPEDEF STRUCT Stm-Tree
      //Definition of the parse-tree for statement (T)
      //generated by C-to-Thumb compiler
   {
      instuction_name : string
      codesize : integer
      NextLink : Stm-Tree
   }


Procedure StmSize(Stm: Stm-Tree):integer
      //It computes the code size of the parse-tree
      //which are stored in Stm.

   Stm-ptr : Stm-Tree
   Stm-ptr ← Stm

   while Stm-ptr ≠ NULL
      StmSize ← StmSize + Stm-ptr.codesize
      Stm-ptr ← Stm-ptr.NextLink
   end-while

   return StmSize
end-procedure
```

Figure 4.13: The algorithm to compute the code size of the single high-level statement.

in Figure 4.13, to compute the object code size for every single statement. We pass the abstract syntax trees that we want to compute into procedure *StmSize*, and it returns the assembly code size.

We make an experiment for the procedural abstraction criteria algorithm. The experiment includes ten C programs from the GNU software[GNUh]. We find out the time of repetition for each program and transform them into the procedural abstraction mode. We use the C-to-Thumb compiler to compile the C programs in PA mode and inline mode respectively, and compute the object code size and the compression-ratio for each program, shown in Figure 4.14. The compression-ratio of the programs are about from 0.8 to 0.9. We note the results of the experiment. If We can find out the the procedural abstraction criteria at high-level stage and apply to code optimization, it may take a good effect for reducing the object code size.

	Prog. Name	Code Fragment	time of repetition	INL	PA	compression ratio
1	lcode.c (lua)	26	2	270	246	0.91
2	adh-opts.c (adns)	98	2	464	384	0.82
3	rsne.c (libf2c)	28	3	558	486	0.87
4	wref.c (libf2c)	44	2	498	456	0.91
5	strtol.c (libiberty)	20	3	286	248	0.86
6	selector.c (libobjc)	128	2	604	514	0.85
7	zutil.c (zlib)	74	2	846	694	0.82
8	minigzip.c (zlib)	156	2	742	630	0.85
9	inftrees.c (zlib)	64	2	302	242	0.8
10	adler32.c (zlib)	172	2	816	742	0.91
Average						0.86

Figure 4.14: Benchmarks from the experiments on the PA criteria algorithm.

Chapter 5

Conclusions

In this thesis, we propose a procedural abstraction model for deciding whether a procedural abstraction transformation on the program reduces the size of object code. In the previous work, the investigation on procedural abstraction has been focused on the applications at the low-level stage. Our procedural abstraction model does not only work at the low-level stage, but also can be extended to the applications at the high-level stage.

We propose the notion of *procedural abstraction criteria*. For the program of consideration, we compute its *procedural abstraction key* k according to the size of a candidate code fragment. Such that, if the candidate code fragment appears more than k times, the procedural abstraction transformation on the program will reduce the size of object code.

In addition, we propose a tree-based approach for computing the procedural abstraction criteria at the high-level stage, called *tree-based procedural abstraction criteria*. We estimate, based on the code generation module, the

size of the object code generated from the AST of the C statements so that our procedural abstraction model may decide whether the procedural abstraction transformation reduces the size of the object code.

We implement the experiments of the procedural abstraction using a simple C-to-Thumb compiler as an evidence that our approach of procedural abstraction criteria can applied to the code compression at the high-level stage.

Bibliography

[ACAP00] G. Araujo, P. Centoducatte, R. Azevedo, and R. Pannain. "Expression-Tree-Based Algorithms for Code Compression on Embedded RISC Architectures," *IEEE Transactions on Very Large Scale Integration(VLSI) Systems, VOL. 8, NO. 5, October 2000.*

[ACCP98] G. Araujo, P. Centoducatte, M. Cortes, and R. Pannain. "Code Compression Based on Operand Factorization," *31st Annual ACM/IEEE International Symposium on Microarchitecture*, 1998.

[AG97] A. W. Appel, and M. Ginsburg. "Modern Compiler Implementation in C," *Cambridge University Press*, 1997, (also available at : http://www.cs.princeton.edu/ appel/modern/).

[Ant97] G. Antoshenkov. "Dictionary-based order-preserving string compression," *The VLDB Journal-The International Journal on Very Large Data Bases*,Volume 6, Issue 1 February 1997.

[ARM00a] "*ARM*^{R} Developer Suite Version 1.1 Assembler Guide," *ARM Limited.*, 2000.

[ARM00b] "ARM Architecture Reference Manual," *ARM Limited.*, 2000.

[ARM98a] "ARM Software Development Toolkit Version 2.50 User Guide," *ARM Software Development Toolkit*, 1998.

[ARM98b] "ARM Software Development Toolkit Version 2.50 Reference Guide," *ARM Software Development Toolkit*, 1998.

[ARMh] "The ARM Ltd. Homepage", *http://www.arm.com.*

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Press, 1986.

[BWC89] T. Bell, I. H. Witten, and J. G. Cleary . "Modeling for text compression," *ACM Computing Surveys (CSUR)*, Volume 21 Issue 4, December 1989.

[BZIP2h] "The former bzip2 and libbzip2 home page", *http://www.bzip2.com/.*

[Cam88] R. D. Cameron. "Source Encoding Using Syntactic Information Source Models," *IEEE Transactions on Information Theory*, Vol. 34, No. 4, July 1988.

[CM99] K. D. Cooper, and N. McIntosh. "Enhanced Code Compression for Embedded RISC Processors," *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, 1999.

[DEM99] S. Debray, W. Evans, and R. Muth. "Compiler techniques for code compression," *Proceeding of the Workshop on Compiler Support for System Software (WCSSS)*, 1999.

[DEMS00] S. K. Debray, W. Evans, R. Muth, and B. D. Sutter. "Compiler Techniques for Code Compaction," *ACM Transactions on Programming Languages and Systems*, Vol. 22, No. 2, 2000.

[Duf52] D. Huffman. "A method for the construction of minimum redundancy codes," *Proc. of IRE, 40 pp.1098-1101* , 1952

[EEFLP97] J. Ernst, W. Evans, C. W. Fraser, S. Lucco, and T. A. Proebsting. "Code Compression," *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1997.

[EF01] W. S. Evans, and C. W. Fraser. "Bytecode Compression via Profiled Grammar Rewriting," *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation*, 2001.

[Eva98] W. S. Evans. "Compression via guided parsing," *Proc. Data Compression Conference (poster session), p.544. 1998,* (also available at : http://www.cs.arizona.edu/people/will/papers/).

[FH95] C. W. Fraser, and D. R. Hanson "A Retargetable C Compiler: Design and Implementation," *Addison-Wesley, Reading, Mass.*, 1995.

[FHP92] C. W. Fraser, D. R. Hanson, and T. A. Proebsting. "Engineering a Simple, Efficient Code-Generator Generator," *ACM Letters on Programming Languages and Systems, Vol. 1, No. 3, September 1992, Pages 213-226*

[FMW84] C. W. Fraser, E. W. Myers, and A. L. Wendt. "Analyzing and Compressing Assembly Code," *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, 1984.

[Fra99] C. W. Fraser. "Automatic Inference of Models for Statistical Code Compression," *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp.242-246.*, May 1999.

[GCCh] "GCC *(GNU Compiler Collection)* Homepage", *http://gcc.gnu.org/.*

[GNUh] "GNU and the FSF *(Free Software Foundation)* Project web server Homepage", *http://www.gnu.org.*

[GZIPh] M. Adler and J.-l.. Gailly. "The gzip home page", *http://www.gzip.org.*

[Hen90] J. E. Hendrix. *A Small C Compiler*, M&T Publishing, 1990.

[Hol95] Klaus Holtz. "Packet Video Transmission on the Information Superhighway Using Image Content Dependent Autosophy Video Compression," *IS&T's 48th Annual Conference*, 1995.

[KKM97] D. Kirovski, J. Kin , and W. H. Mangione-Smith. "Procedure Based Program Compression," *Proceedings of Microarchitecture*, 1997

[LBCM97] C. Lefurgy, P. Bird, I. C. Chen, and T. Mudge. "Improving Code Density Using Compression Techniques," *Proceedings of the Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture*, December 1997.

[LC02] K. Lin, and C. P. Chung. "Code Compression Technigues Using Operand Field Remapping," *IEE Proceeding E, Computers and Digital Techniques*, Vol. 149, No. 1, 2002.

[Lee85] J. Lee. "ANSI C grammar (Lex)ANSI C grammar, Lex specification," *published the Lex specification together with a Yacc grammar for the April 30, 1985 ANSI C draft.*

[LH87] D. A. Lelewer, and D. S. Hirschberg. "Data Compression," *ACM Computing Surveys (CSUR)* , September 1987 Volume 19 Issue 3.

[LMB92] J. R. Levine, T. Mason, and D. Brown. "Lex & Yacc 2nd Edition," *OŔeilly & Associates, Inc.*, 1992.

[LO95] M. Loukides, and A. Oram. "Programming with GNU Software," *OŔeilly & Associates, Inc.*, 1995.

[LSC02] K. Lin, J. J. Shann , and C. P. Chung. "Code Compression by Register Operand Dependency," *Proceedings of the Sixth Annual*

Workshop on Interaction between Compilers and Computer Architectures (INTERACT.02) , 2002.

[Luc00] S. Lucco. "Split-Stream Dictionary Program Compression," *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, May 2000.

[LW99] H. Lekatsas, and W. Wolf. "SAMC: a code compression algorithm for embedded processors," *IEEE Transactions on CAD*, 1999.

[Muc97] S. S. Muchnick. "Advanced Compiler Design and Implementation," *Morgan Kaufmann Publishers.*, 1997.

[Pra02] S. Prata. "C Primer Plus 4/e," *Sams Publishing.*, 2002.

[RR01] S. P. Reiss, and M. Renieris . "Encoding program executions," *Proceedings of the 23rd international conference on Software engineering*, 2001.

[SB94] A. Silberschatz, and P. B. Galvin. "Operating System Concepts," *Addison-Wesley Publishing Co.*, 1994.

[WC92] A. Wolfe and A. Chain. "Executing compressed programs on an embedded RISC architecture," *Proceeding on the* 25th *Annual International Symposium on Microarchitecture*, 1992.

[WNC87] I. H. Witten, R. M. Neal, and J. G. Cleary. "Arithmetic coding for data compression," *Communications of the ACM June*,Volume 30 Issue 6, 1987

[YSOOS97] Y. Yoshida, B. Y. Song, H. Okuhata, T. Onoye, and I. Shirakawa. "An object code compression approach to embedded processors," *Proceedings of International Symposium on Low Power Electronics and Design*, 1997.

[Yun98] C. Yung. "EAS: An Experimental Applicative Language With Sets," *Proceedings on 1998 MASPLAS (in cooperation with ACM SIGPLAN)*, Rutgers University, New Jersey, April 1998, (also available at : http://bryde.csie.ndhu.edu.tw/yung/index.xml).