Survey of Code Compression

Shin-Lin Chen
{m8921505}@mail.ndhu.edu.tw

August 20, 2002

1 Problem and Motivating

With the present day, such embedded system as PDAs, handhelds, IA devices and personal communicators and so on developed very fast so it's applications became very complex. Relative to its development, its demand for system resources become very large. These system resources include processor caches, memories, local storages and remote storage via network etc. [3] In general, the growth of these resources is slower than need of these. To reduce the system resources in use is a effective method for further the system performances. There are following three main methods to reduce the system resources in use:First, the system reduces as soon as possible the amount of data memories in use at running time, but its over-head may be increased. The second, the code generator generates a compressed code and a number of data as dictionaries[1] etc, so it needs a decompress stage before executed programs at run-time. Third, the code generator of compiler also generates a compressed code called compacted form but it not need decompress at run-time. It can running programs directly and it is a zero-overhead method[3]. This paper now focus on this method.

On the previous researches of code compression has three different ways:hight-level language(source code), IR/byte code and low-level language(assembly/machine code)[6]. On low-level language, the procedural abstraction (PA) is a common method for code compression[6]. Figure.2 shows an example of procedural abstraction using ARM/Thumb instruction. The conception of procedural abstraction derived from *code factoring*[6]. The meaning of code factoring is that identified the *"identical"* sequences of instructions and factors out these sequences of instruction as a single procedure. These procedure are used in place of original code fragments. Procedural Abstraction is a good approach, but it is not effective in any situation. In this paper, we construct a notion about procedural abstraction called *procedural criteria*. It can determined which condition is beneficial for code compression and which condition is without benefit. For Example, if the sequence of instructions are too small, the compiler will avoid procedural abstraction. In section 2, we will give an example to solve the procedural criteria and it focused on low-level language. Section 3 consider procedural criteria at high-level language like C, java etc. , and we implement a simply compiler to generate ARM/Thumb assembly code from ANSI C language. Finally, we discuss the influence of procedural criteria on my implementation.

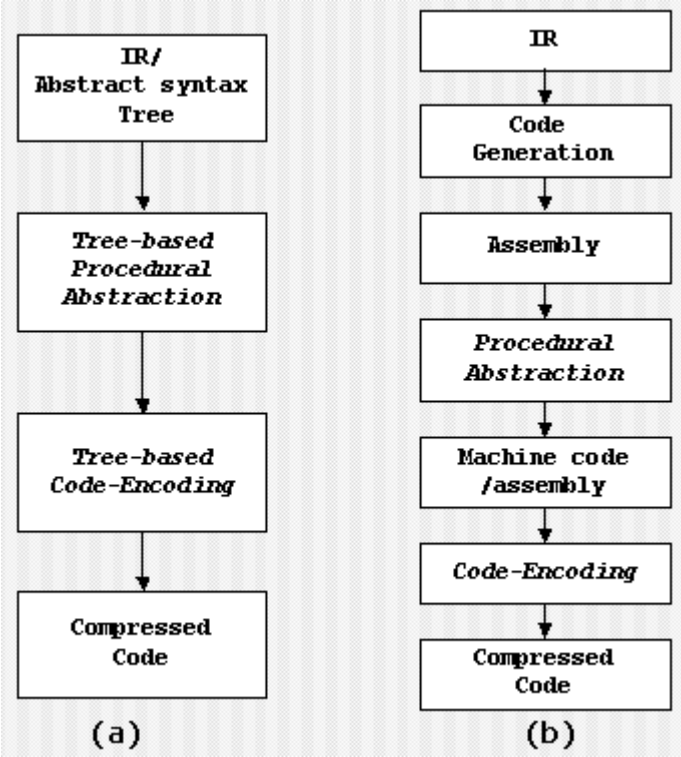

Figure 1: Code compression about this research(a)my research (b)previous research.

```
    .                                    .
    .                                    .
MOV r1,r0                            BL myproc
MOV r2,#8                                .
LSR r0,r1,#28                        BL myproc
    .                                    .
MOV r1,r0                myproc      MOV r1,r0
MOV r2,#8                            MOV r2,#8
LSR r0,r1,#28                        LSR r0,r1,#28
    .                                MOV pc,lr
    .                                    .
    .                                    .
   (a)                                  (b)
```

Figure 2: Code fragment in (a) inline and (b) Procedural Abstraction.

2 Background and Related Works

2.1 Enhanced Code Compression for Embedded RISC Processors

In embedded system, the size of compiled code is increasingly important. This paper explores compiler techniques for reducing memory. It is using pattern *matching* techniques and *repeated instruction sequences*. This approach run program executables directly, without an intervening *decompression stages*.

2.1.1 Introduction

Have many factors determine the size of code:(1)The *instruction set architecture* of the target machine has a strong effect.(2) Specific code sequences selected by the compiler have an effect, as do the specific transformations applied during optimization. This paper explores one technique for reducing code size code compression during the late stages of compilation The approach is conceptually simple it builds on early work by Fraser et al[5] for VAX assembly code. Pattern matching techniques identify identical code sequences *(Repeat)*. Use *procedural abstraction* and *cross-jumping* to channel execution of the repeat through a single copy of the code. This paper also extend this basic algorithm by relaxing the notion of identical to abstract away register names a key enhancement when compressing code compiled with a *graph coloring register allocator*. Finally we describe a profile based technique that provides a mechanism for controlling tradeoffs between code size and overall execution time.

2.1.2 Identifying Repeats

The first task in our compression framework is to *find* all the repeats in the program and to select a set of in stances to be compressed. To identify repeats it builds a suffix tree as in the work of Fraser et al[5]

Suffix Tree Construction A suffix tree is a data structure Encodes information about repetition within a textual string. Suffix tree are used for a variety of a pattern matching application. Figure shows an example suffix tree for the text string bananas.

Next, it hash each instruction and enter it into a global table. Each instruction with a unique combination of opcode, registers and constant receives its own table entry. For example : iADD r9,r10-¿ r10, iADD r10,r9-¿ r10. We create a linear, string-like representation of the program called the *text*. Each character corresponds to a particular instruction in the program.

Building the Repeat Table We store information gleaned from the suffix tree in a data structure called the repeat table. Each entry in the table a repeat is composed of a set of fragments or specific identical substrings within the program text. Figure gives an example of a repeat consisting of two identical fragments. After a set of fragments has been collected into a repeat, the compiler must analyze them to identify any conditions that would inhibit the transformations. We refer to these conditions as control hazards, they correspond to jumps into or out of the fragment that interfere with *procedural abstraction* or *cross jumping*.

Repeat Splitting When the repeat manager identifies a hazard that prevents some or all transformations within a repeat it typically handles the situation by splitting the repeat. Given a repeat with N fragments splitting partitions each fragment at a specific offset creating two new repeats each with N smaller fragments. For example in Figure the compressor could eliminate the control hazard in the second fragment by splitting the entire repeat at o set producing two new repeats each with two fragments.

The repeat manager uses a detailed cost model to decide where and when to split a given repeat, taking into account the fragment length the offset and type of the hazard and the number of fragments that exhibit the hazard.

2.1.3 Replacing Repeats

Our framework uses two distinct transformations to achieve this procedural abstraction and cross jumping. Figure shows an example of procedural abstraction. In this transformation a given code region is made into a procedure and other regions identical to it are replaced with calls to the new procedure. Procedural abstraction requires that the candidate regions be single entry single exit internal jumps must be within the body of the region. Figure shows an example of cross jumping some times known as tail merging, in which identical regions that end with a jump to the same target are merged together. In this transformation we replace a region with a direct jump to another identical region. All of the out branches in each region must match in order for cross jumping to be applied.

Both these transformations have certain costs both in terms of code space and execution time. For example when forming an abstract procedure the compiler must add a return instruction at the end of the body and must insert a call instruction at each place where the abstract procedure is referenced.

The default strategy for deciding which repeats to com press is very simple the compiler calculates expected savings for each repeat sorts the repeat table by expected savings and applies transformations in sorted order. Since repeats can overlap, the compiler must take care to avoid compressing a given code fragment more than once.

2.1.4 Extension

Abstracting Branches The first step in relaxing our notion of identity is to rewrite branches into a pc relative form whenever possible as in the work of Fraser et al. Recoding these branches into a pc relative form allows the suffix tree construction algorithm to discover repeats that span multiple blocks. After rewriting it finds many cross block repeats.

Abstracting Registers Often two code fragments are identical except for minor differences in register use. Consider the example in Figure . These two fragments have identical opcodes, but they employ different registers wherever the first fragment uses r7 the second fragment uses r6 and vice versa.

Relative register pattern matching When an instruction I reads register rk we look for a previous reference to rk or definition of rk within the current basic block. If a previous reference to rk exists in instruction Q, then we rewrite the reference to rk within I as a tuple ¡O,T,R¿. If no previous reference to a register exists, then we rewrite the reference as a placeholder or wildcard token. All de definitions within the block are rewritten as.

Figure shows one of the code fragments from Figure along with the same fragment after register references are rewritten as relative offsets.

Register renaming When the suffix tree is built based on the rewritten representation of the input program two fragments may be placed into the same repeat, even though they use different registers. Thus to compress the fragments together, we may need to apply register renaming to make one fragment identical to the other. The renaming strategy is called live-range recoloring. Showed in Figure.

Profile-based selection For each function F the compiler computes the ratio of F s dynamic instruction count to F s static instruction count we call this ratio RF Given a total of N functions, the compiler then selects the Qth order statistic from among all the N ratios the default value for Q is we call this selection the *cutoff ratio.*

2.1.5 Summary

In this paper we have described and evaluated new ex tensions to suffix tree based code compression showing how a compiler can use them to produce smaller more compact code while still retaining a directly executable program.

Our data show that relaxed pattern matching in combination with live range recoloring improves the effectiveness of code compression substantially in creasing the average code space reduction from around 1% to 5% just under for the benchmarks we studied.

2.2 Compiler Techniques for Code Compaction

2.2.1 Introduction

3 Idea

On previous researches for code compression about procedural abstraction and code encoding, it is focused on low-level language such as assembly, it showed in Figure (b). Now, my method focus on high-level language. We use compiler techniques to implement procedural abstraction and code encoding showed in Figure (a). To test and verify whether procedural abstraction and code encoding can apply to high-level language or not.

My method is to implement a simple compiler back-end and code-generation. This simple compiler use traditional compiler front-end tools such as lex and yacc, it translates the high-level language to inter-media representation (IR), and we construct a code generation that it translated IR to assembly codes(ARM/Thumb assembly codes) . Finally, we use both the source codes and assembly codes to test and verify the procedural abstraction and code encoding can applied to high-level language.

4 Plan and Schedule

	Date	content
1.	2002/7/1 2002/8/31	Implement and setup environment of research(C-to-Thumb compiler)
2.	2002/8/1 2002/9/31	Implementation of *Procedural abstraction* (identify Repeat or not?)
3.	2002/10/1 2002/10/31	Implementation of *Procedural abstraction* (High-Level Language)
4.	2002/10/1 2002/12/31	Implementation and Research of *Field Encoding*
		discuss with professor Yung. about this schedule
5.	2002/9/1 2002/12/31	to test and verify between implementation and Theory
6.	2002/10/1 2003/1/31	Analysis of Theory and Methodology

References

[1] Charles Lefurgy, Peter Bird, I-Cheng Chen, and Trevor Mudge. "Improving Code Density Using Compression Techniques." *Proceedings of the thirtieth annual IEEE/ACM international symposium on Microarchitecture December* , 1997.

[2] K.Lin and C.-P.Chung. "Code compression technigues using operand field remapping." *IEE Proc.-Comput.Digit.Tech,Vol 149,No1* , 2002.

[3] William S. Evans and Christopher W. Fraser. "Bytecode Compression via Profiled Grammar Rewriting." *ACM SIGPLAN Notices , Proceedings of the ACM SIGPLAN'01 conference on Programming language design and implementation*, 2001.

[4] Keith D. Cooper and Nathaniel McIntosh. "Enhanced Code Compression for Embedded RISC Processors." *ACM SIGPLAN Notices , Proceedings of the ACM SIGPLAN '99 conference on Programming language design and implementation*, 1999.

[5] Christopher W. Fraser,Eugene W. Myers and Alan L. Wendt. "Analyzing and Compressing Assembly Code." *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction SIGPLAN Notice8 VoL 19, No. 8* , 1984.

[6] Saumya K. Debray,William Evans,Robert Muth and Bjorn De Sutter. "Compiler Techniques for Code Compaction." *ACM Transactions on Programming Languages and Systems, Vol. 22, No. 2.*, 2000.

[7] Aho, Sethi and Ullman. *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Press, 1986.

[8] Andrew W Appel and Maia Ginsburg. "Modern Compiler Implementation in C." *Cambridge University Press*,1997.

[9] Jens Ernst,William Evans,Christopher W. Fraser,Steven Lucco and Todd A. Proebsting. "Code Compression." *ACM SIGPLAN Notices,Proceedings of the 1997 ACM SIGPLAN conference on Programming language design and implementation.*, 1997.

[10] "The ARM Ltd. HomePage", *http://www.arm.com* .

[11] James E.Hendrix. "A Small C Compiler." *M&T Publishing,Inc.*,1990.