Code-Size Reduction for Embedded Systems using Bytecode Translation Unit

Phanupan Nanthanavoot
Department of Computer Engineering,
Faculty of Engineering, Chulalongkorn University
Phyathai Road, Pratumwan Bangkok 10331
phanupan.n@cp.eng.chula.ac.th

Prabhas Chongstitvatana
Department of Computer Engineering,
Faculty of Engineering, Chulalongkorn University,
Phyathai Road, Bangkok 10330, Thailand.
prabhas@chula.ac.th

ABTRACT

This work introduces a technique which applies a stack-based intermediate code, also called as bytecodes, to reduce the size of programs in an embedded system. A hardware interpreter known as the *Translation Unit* translates bytecodes into native codes before execution. Experiments show that a program written in bytecodes is smaller than one written in native codes by 16%-38%.

Keyword: Code-size reduction, Code compression, Embedded system, Bytecode.

1. INTRODUCTION

Embedded microcontrollers are highly constrained in cost, power and area. Therefore it is important to reduce the microcontroller die area. This will increase the amount of die per wafer and eventually increases the die yield in the microcontroller production. In addition, decreasing the size of program memory, a major part of the embedded microcontroller, will also reduce the die area for on-chip memory.

This work introduces a way to run small-sized programs in an embedded system using a combination of interpreter and stack-based intermediate codes, or also called bytecodes, which will reduce the program size.

This paper is sectioned into 5 parts. Section 2, the fundamentals of code size reduction and its efficiency metrics are explained. The next section discusses the proposed technique. In Section 4, the experiments to measure the compression ratio and the result are shown. After the research summary in Section 5, the last section details the related work of code size reduction.

2. CODE-SIZE REDUCTION BACKGROUND

Code size reduction is a technique to reduce code size. There are two popular techniques: code compaction and code compression.

The first technique, code compression, uses data compression algorithms on machine codes. On the other hand, code compaction reduces the program size by using compiler optimization to rearrange and eliminate superfluous codes. This allows the compressed program to be executed immediately without needing decompression as in the code compression technique. Decompression will show down the system operation in code compression. However, using compression algorithm in code compression will reduce the program size more than using code compaction.

The efficiency of the code size reduction technique is measured through the compression ratio as in equation (1)

$$\text{Compression Ratio} = \frac{\text{Compressed size}}{\text{Uncompressed size}} \qquad (1)$$

3. SYSTEM DESIGN

3.1 Overview

This work introduces a way to reduce the program size using the approach in [1, 2] which says that a program in the form of the intermediate code of a stack-based instruction set will be smaller than a program in the form of the machine code of a register-based instruction set. An example of a popular bytecode is the Class File or Java bytecodes [3] of the Java language.

The reason a program in the form of the bytecode is smaller than one in the form of the machine code is as follow:

- Bytecode instruction set has higher semantic content than register instruction set. Therefore, a bytecode instruction is equal to many register machine instructions.
- Bytecode is a stack-based instruction set which the location of an operand is implicit in the stack pointer. On the other hand, the operand of a register machine must be declared explicitly, so bytecode instruction's size is smaller than register instruction's size.

There are two alternatives to implement bytecodes in an embedded system. The first alternative is to build a machine that can execute bytecodes directly. The machine of this type is called a stack machine.

The second alternative is to run bytecodes on a virtual machine. The virtual machine can be hosted on any architecture. The popular choice is to host a virtual machine on a register-based machine because of the availability of high performance register-based processors in the market.

The virtual machine uses an interpreter to translate the bytecode instruction into the register-based instruction. One of the most time consuming operation in interpreting a bytecode is the instruction dispatch. The dispatcher in a high-level language implementation of a

virtual machine is composed of a switch-case construct for each bytecode instruction. This causes the operation to be slower than the operation of the native code.

3.2 Design

To improve the speed of execution of bytecodes, a hardware virtual machine is used. The hardware interpreter is shown in Figure 1. A register-based processor core is assumed. The translation unit is the main contribution of this work. The details of this unit are discussed next.

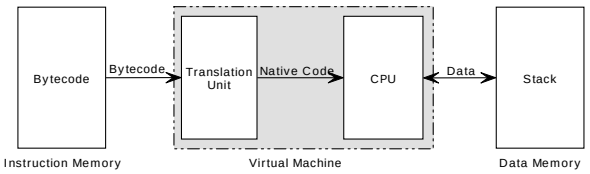

Fig. 1: *Virtual machine with translation unit*

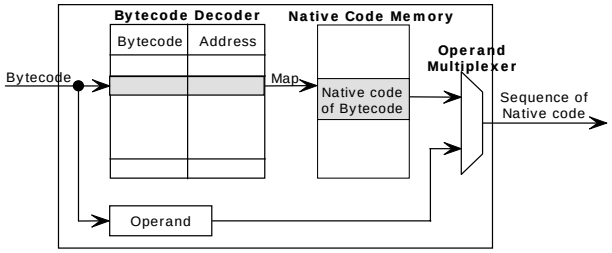

Fig. 2: *Translation unit component*

The components of the translation unit are shown in Figure 2. They consist of the bytecode decoder, the native code memory and the operand multiplexer. In each operation of bytecode, the native code memory records the sequence of native codes which achieve the correct operation.

The bytecode decoder is a look-up table that stores the address and the number of native codes in the sequence. It maps a bytecode into the sequence of native codes in the native code memory.

Some bytecode contains an operand such as a literal (Figure 3), an instruction that pushes an immediate operand into the stack. In the operation of a virtual machine, the operand in the bytecode must be passed to the operand field of the correct native code in the sequence. The operand multiplexer in the translation unit will send the operand to the first instruction of the native code instruction which allows the CPU to read the operand from the bytecode.

For embedded system applications, one major consideration is the circuit size of the translation unit. The size of the translation unit depends on the size of the look-up table in the decoder and the size of the native code memory in the translation unit. The size of the look-up table depends on the amount of entry or the number of bytecode instructions in the table. For the native code memory, its size depends on the length of the sequence of native codes corresponded to a bytecode. This is affected by the difference of the bytecode instruction and the architecture of the CPU.

Consequently, the bytecode instruction set should not include too many instructions. This work employs 27 simple bytecode instructions from [4]. A small size CPU, suitable for an embedded system in [5, 6], is used. The CPU consists of 4 registers:

- Stack Pointer (SP) which points to the data on the top of stack,
- Frame Pointer (FP) which manages subroutine calls,
- Top of Stack (TOS) which caches the topmost value of the stack in the register, and
- Buffer (BUFF) which keeps intermediate values.

The translation unit fetches bytecodes from the instruction memory and feeds CPU with native codes. Because the addresses of bytecode are different from the addresses of native code, the control flow instructions such as jumps and calls require special attention. The translation unit feeds the native jump instruction to the CPU so that the program counter points to the appropriate bytecode. For the call instruction, the CPU performs save/restore the program counter to the stack segment. The translation unit must feed the correct sequence of native codes to achieve this effect.

An example of translating a bytecode to the native code is the translation of the ***Literal*** instruction that pushes a constant into the top of stack and the ***Add*** instruction that adds 2 top values in the stack and keeps the result in the top of stack are shown in Figure 3.

Bytecode	Native code
Literal #constant	**movi** buff, #constant **stw** tos, 0(sp) **mov** tos, buff **subi** sp, 1
Add	**ldw** buff, 1(sp) **add** buff, tos **mov** tos, buff **addi** sp, 1

Fig. 3: *Example of bytecode translation : Literal and Add instructions*

The system is developed in the form of RTL (Register Transfer Level) using Verilog HDL. It is verified by simulation method through the program ModelSim version 5.6e, Xilinx.

4. EXPERIMENT

The purpose of the experiment is to measure the efficiency of code size reduction. The compression ratio is measured using the integer benchmark Stanford (Hennessy and Nye). The description of each program in the benchmark is shown in the following Table 1.

The size of the program compiled in the bytecode compared with a program in the native code. A special compiler is used to compile high-level programs into bytecodes. A simple instruction specialization is applied to the bytecode programs. The frequently used sequences

of bytecodes in the program are replaced with a special instruction to reduce the size. The lists of the special instructions are shown in table2.

Table. 1: *Stanford benchmark*

Benchmark	Description
Bubble	Sort 20 numbers by bubble sort algorith
Quick	Sort 20 numbers by quick sort algorithm
Hanoi	Find a solution to move 3 disks in problem - tower of hanoi
Sieve	Find all prime numbers less than 100
8-Queen	Find all solutions of 8-queen problem
Matmul	Multiply matrix 5×5
Perm	Permute 4 digits of 0, 1, 2, 3

A special compiler is used to compile high-level programs into bytecodes. The size of a program compiled into bytecodes is compared with the program in the native code. A simple instruction specialization is applied to the bytecode programs. The frequently used sequences of bytecodes in the program are replaced with a special instruction to reduce the size. The lists of the special instructions are shown in Table2.

Table. 2: *Special bytecode instructions which are added into the bytecode instruction set*

Bytecode instruction	Function
INC #local	Increment the local variable
DEC #local	Decrement the local variable
Lit0	Push literal 0 to the top of stack
Lit1	Push literal 1 to the top of stack
Rval1, Rval2, Rval3, Rval4	Get local variable 1, 2, 3 or 4 and push it into the top of stack
JLt #address	Jump if the top of stack is less than the second
JEq #address	Jump if the top of stack equals the second

Table. 3: *Size's comparison between bytecode and native code program (in bytes)*

Program	Bytecode size	Native code size	Compression Ratio
Bubble	128	158	0.81
Quick	253	306	0.82
Hanoi	128	178	0.71
Sieve	154	196	0.79
8-Queen	125	168	0.74
Matmul	253	298	0.84
Perm	221	356	0.62

The programs in native code are written in an assembly code. They are directly translated from the high-level code. Register allocation is not applied in the translation as there are only 4 registers. The size of the program in bytecode and native code are shown in Table 3.

5. CONCLUSION

The result in the experiments shows that the compression ratio is ranged from 0.60 to 0.84 with an average 0.76. It still can be reduced further through sequence analysis of the common bytecode and substituting those redundant sequences with a special instruction.

Presently, the system has been tested on a simulator. The next step is to develop the system to operate on a real chip using the FPGA (Field Programmable Gate Array) technology. The circuit size of the translation unit can be assessed.

6. RELATED WORK

Thumb [7] and MIPS16 [8] are designed to decrease program size by redesigning instruction set of the processor ARM and MIPS which are 32-bit RISC processors to 16-bit instruction sets. These new instruction sets are able to work compatibly with the original processor cores. Compression ratios of both works are 0.70 and 0.60 respectively.

Code compression for RISC Processor (CCRP) [9] introduces a method to compress a program using Huffman algorithm to compress code and cache memory. The cache memory stores an instruction before it is used by the processor unit. The compression ratio of this work is 0.73.

Lefurgy [10] observed the compiler's method of translation and found that some sequences of instructions are redundant. Therefore those repetitions are replaced with codewords which used fewer bits. These codeword are stored in a dictionary. When the processor executes a codeword, the decompressor will retrieve the sequence from the dictionary. The experiments are performed on 3 types of processors: PowerPC, ARM and i386. The compression ratios of each processor are 0.61, 0.66 and 0.74 respectively.

IBM uses the technique, called "CodePack" [11, 12], to compress the program in PowerPC. It applied two compression concepts: dictionary compression in [10] and decompression on the cache in [9]. Compression ratio of this work is 0.60. However in [13] the reported performance of the CodePack system is that it is slowing down the operation by 0.14-0.18 times.

Ernst [14] introduced BRISC based on two concepts operand specialization and opcode combination. BRISC is implemented as an interpreter. The result of the experiment showed 0.53-0.69 compression ratio. However, the interpreter slowed down the system by 9.6-15.4 times compared to the execution of the uncompressed code.

These works demonstrate the effectiveness of code-size reduction using various schemes of code compression and compiler optimizations. The down side is the run-time overhead associated with the interpreter. The translation unit proposed in this paper should prove to be effective in terms of small run-time overhead. The compressed ratio achieved by the proposed method is comparable to the existing methods.

REFERENCE

[1] Chongstitvatana, P. The art of instruction set design. In Conference of Electrical Engineering, Thailand, 2003.

[2] Koopman, P. STACK COMPUTER, the new wave. Ellis Horwood, 1989

[3] Joy, B., Staddle, G., Gosling, J. and Bracha, G. JAVA™ Language Specification (2nd edition), Addison Wesley Pub, 2000

[4] Chongstitvatana, P. Final Report : A multi-tasking environment for real-time control [online]. 1998. Available from: http://www.cp.eng.chula.ac.th/~piak/r1/final.pdf [November 2003].

[5] Piromsopa, K. Development of A Reconfigurable Embedded Web Server. Master Thesis, Chulalongkorn University, 2000.

[6] Bavonparadon, P. and Chongstitvatana, P. RTL formal verification of embedded processors. In IEEE International Conference on Industrial Technology, pp. 667-672. 2002.

[7] Advanced RISC Machines Ltd. An Introduction to Thumb. March 1995.

[8] Kissell, K. D. MIPS16: High-density MIPS for the Embedded Market. In Proceedings of Real Time Systems '97 (RTS97), 1997.

[9] Kozuch, M. and Wolfe, A. Compression of embedded system programs. In Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors. IEEE Computer Society Press, Los Alamitos, Calif. 1994

[10] Lefurgy, C., Bird, P., Chen, I., and Mudge, T. Improving code density using compression techniques. In International Symposium on Microarchitecture 30 (1997).

[11] IBM. CodePack PowerPC Code Compression Utility User's Manual Version 3.0. IBM, 1998.

[12] Game, M. and Booker, A., CodePack: Code Compression for PowerPC Processors. MicroNews 5 (1) , IBM, 1999.

[13] Lefurgy, C., Piccininni, E., and Mudge, T. Evaluation of a high performance code compression method.Proceedings Annual International Symposium on Microarchitecture 32nd (1999) : 93-102

[14] Ernst, I., Evans, W., Fraser, C. W., Lucco, S. and Proebsting, T. A. Code compression. Proceedings of the ACM SIGPLAN'97 Conference on Programming Languages Design and Implementation 32 (15 – 18 June 1997) : 358 - 365