Nibbling Java Byte Code for Resource-Critical Devices

Vishnu Kotrajaras and Prabhas Chongstitvatana

Department of computer engineering
Chulalongkorn University
Thailand
vk1@cp.eng.chula.ac.th, prabhas@chula.ac.th

Abstract

This work describes a method that makes the Java instruction as small as possible. The proposed method, called "nibble coding", compresses two instructions into one byte. The experiment is carried out to compare Java byte code instruction and a modified code with the nibble coding. The result shows the proposed scheme achieves a much smaller instruction bandwidth than the ordinary byte-code counterpart.

Key-Words: *code compression, instruction bandwidth, Java byte code*

Introduction

The progress of microprocessor design has been phenomenal in the last decade. The performance is doubled every 18 months, as stated by Moore's law. In the last decade, drive for performance is the key for progress. In time, the market competition forces the architecture to slowly converge and Instruction Set Architecture (ISA) has become not as important as in the previous decade where the market was still young. However, new applications have shifted the design landscape once again to low power, portable devices [1]. The technology for implementing computational devices in small batch with fast turnaround time has opened up the issue of ISA design.

For some very small devices, the instruction storage can be absolutely critical. This work describes a new investigation of the old problem: How to make the instruction (program) as small as possible. We experimented with Java and proposed to compress two instructions into one byte. This scheme is called "nibble coding". The investigation is carried out to compare the instruction bandwidth using the measure of dynamic instruction count on a small suite of benchmark programs. The result shows that this scheme achieves a much smaller code size. Reducing the size of code has benefit in two aspects. The first one is obvious in reducing the storage requirement, both in code segment and in instruction cache memory. This is often the reason behind many classic instruction set architectures, to achieve very compact executable code. The second one is related to power requirement. As the instruction bandwidth is reduced, the power consumption is also reduced [2, 3].

Choosing code to compress

Conventional machine code is not the most compact form to represent an executable code. The intermediate code for a virtual machine is usually much smaller because of its higher semantic content. One of the most popular form of intermediate code is based on stack addressing. A stack machine code is very compact due to its use of stack, which does not require addressing bit. Majority of instructions thus do not require the operand in the instruction as it is implicit in the stack. Basically the stack instruction has two forms: zero argument and one argument. All arithmetic and logic instructions are zero argument. The jump/call load/store instructions have one argument, the address of jump or the data memory. The most well-known stack virtual machine is Java Virtual Machine (JVM) [4,5,6] as it is embedded into most browser. Its machine code is called byte code. As the name implied, the format of code is byte oriented where most instruction is one byte. We chose Java byte code for our experiment because of its small size as a stack machine code and because of its popularity.

Code compression

There are a number of work in code compression [7, 8]. This work differs in that it is concentrated on virtual machine code, specifically Java byte code. There are many existing methods to pack instructions into a smaller space, such as:

1. Extended instruction, making a special instruction that replaces several simple instructions.
2. Specialization, eliminate argument by making an instruction special to a particular argument thus reduce the size of instruction.
3. Reduce the size of argument, by using a literal table that store a number of full-size arguments. The argument of the instruction can be replaced by the index into this table. The index is much smaller than full-range of argument as there are small finite number of different variables in a program.

The contribution of this work is the technique of packing two instructions into one instruction, or "nibble coding".

Extending instruction set is a very powerful method and has potential to reduce the size of code beyond what achievable by other methods. It has been explored in our previous work [9]. For example, the expression `i = i + 1` which can be translated into the following sequence of stack code: `address of i, value of i, literal 1, plus, store,` totally five instructions can be replaced by one special instruction: `increment i`. However, beside some obvious idiom, selection of special instructions faces combinatorial explosion as the combination of code grows very fast with the length of sequence. Another limitation of this technique is that except combination of length two, it is applicable only to a small percentage of the whole program.

To specialize instructions the frequency of use of each instruction including its argument is measured. A number of most often used instructions are then made into special instructions with no argument. This method eliminates the space to store argument completely.

Using a table to store literals in a program can save large amount of bit in instructions that are required to store full-size literals such as the address of a variable. An index into this table is used as argument instead. The size of this index depends on the size of the table. A careful judgement is required to balance the size of the table to cover large number of literals appeared in a program without making the size of index too large.

Nibble coding is the main technique to encode two instructions into one. The instruction space is divided into normal instructions and packed instructions. For example, in byte code format, the normal instructions occupy half of 256 instructions and the rest is for packed instructions. The space for instruction encoding is limited to half-length of the normal instruction, for example 7 bits is remained to pack two instructions in the byte-code format. The choice of two-instruction combination is based on:

- The frequency of use, by compacting the most used combination, the impact in dynamic code compression is maximized.
- The frequency of occurrence, the combination that appears most frequently in the program will reduce the static code size.

The nibble coding can be designed to be orthogonal, that is, it can be full combination of the selected instructions. In this aspect, it can be applied to a large percentage of code sequence. This is in contrast to the extended instruction concept mentioned previously. The result of this technique can achieve 50% code size reduction in average when the selection of instruction covers the program well.

To investigate the idea, the experiment of application of these techniques is carried out on the Java virtual machine. Various effects are measured to ascertain their actual contribution to code compression. The detail of which will be describe in the next section.

Experiment

A small benchmark suite to test integer instructions [10] is used to collect statistic of behavior of code execution. The benchmark suite consists of seven programs:

1. **Sieve:** test normal loop. Sieve prime number, the method of Erathothenes, find the prime <= 100.
2. **Hanoi:** test recursion. Move 6 disks from peg 1 to peg 3.
3. **Matmul:** test loop and arithmetic. Multiply 4 by 4 matrix, C = A * B.
4. **Bubble:** test loop and swap. Bubble sort, with input data 20..1.

5. **Qsort:** test loop and recursion. Quick sort, with input data 20..1.
6. **Perm:** test recursion. Permutation generator. Permute 4 numbers: 0 1 2 3.
7. **Queen:** test loop and index. Find all solutions of 8-queen problem encoding the solution as column position {0, 1, 2, 3, 4, 5, 6, 7}

The static size of all programs and the dynamic size of executing these programs are collected and are used to compare with the proposed code compression method.

Profiling the benchmark suite results in the following general observation:

- The top 10 most often used instructions consume 96.5% of instruction bandwidth (Table 1). The next four instructions each consume only 0.5% of the bandwidth, and the rest each even consume much less bandwidth.
- The most often used instruction is "getstatic" (loading value of class variable to stack).
- The literal 0 and literal 1 constitute almost 100% of all literals executed.

getstatic	18.95886
iload_1	18.81711
iload_0	13.55789
if_icmplt	10.61253
iadd	9.089021
iconst_1	6.889043
iaload	6.346258
istore_1	4.574203
isub	3.884542
iastore	3.762348

Table 1: the top 10 most often used instructions (percent)

Nibble coding

The constraints in our code compression are as follows:

- For byte code, the natural boundary for accessing a code is a byte, the encoding of our code compression will follow this byte addressing.
- Sequence of code in consideration is in a basic block, although compressing combination across jump is possible, it is not attempted in this work.

Using 8-bit for an instruction, the instruction space is 256 instructions. The first half is allocated to normal instructions and special instructions. The second half is reserved for nibble coding. There is 7-bit space of which will be divided into 3 and 4-bit for the first nibble and the second nibble following the observation from the code execution profile that the leading instruction of 2-combination is more constrained than the followed instruction. The basic block constraints that the leading instruction cannot be the control flow (such as jumps and method calls, for example: `if_icmplt, if_icmpne, ifeq, return, invokevirtual, invokespecial`). Another constraint is that instructions in the nibble can not both have arguments at the same time otherwise argument encoding will be complicated and will not be compact. To consider the selection of instruction of nibble coding, the lead and follow instruction sets will be considered separately.

To form a set of 128 compressed instructions, eight instructions are chosen to be the lead instructions according to their frequency of use and frequency of instruction pairs appearing in our benchmark programs. They are { `getstatic, iload_0, iload_1, iconst_1, iaload, istore_1, iconst_0, iadd` }. The follow set can have 16 instructions and the choice is the lead set plus { `isub, if_icmplt, return, goto, iastore, iload_2, invokestatic, iload_3` }.

As only argument from one instruction is allowed in nibble coding, there are instructions that cannot be compressed with others, such as `if_icmpne` and `if_icmpge`. Although such instructions have higher occurrences than `iload_3`, we have to remove them from our list.

The normal instructions occupy the first half of instruction space and are begun with a bit 0. The second half is for nibble code. The nibble code starts with a bit 1, the next 3-bit specifies the lead instruction, the last 4-bit specifies the follow instruction. If the instruction has argument, then the nibble has one byte argument. (Fig 1)

Normal instructions

```
0|op:7             ;; zero argument
0|op:7 a2:8        ;; one argument
0|op:7|a1:8  a2:8  ;; two arguments
```

nibble instructions

```
1|op1:3|op2:4        ;; zero argument
1|op1:3|op2:4 a2:8   ;; one argument
```

Figure 1: Instruction encoding of the proposed scheme

Result and discussion

Table 3 shows all the raw data on static and dynamic measures of the execution of benchmark programs. Figure 2 shows the improvement of code size reduction. From Fig. 2, the nibble coding affects on average 30% of static code size, with the maximum reduction of 38% for the permutation program and the minimum reduction of 23% for the quicksort program. For the dynamic code size, the average reduction is 36%, with the maximum reduction of 50% for the 8-queen program and the minimum reduction of 25% for the quicksort program. Comparing with a typical 32-bit 3-address machine code of a load/store register processor, [11], the proposed method achieves the static code size of 74% less and the dynamic code size of 70% less.

References

[1] C. Kozyrakis and D. Patterson, "A new direction for computer architecture research", IEEE computer, Nov. 1998, pp.24-32.

[2] R. Gonzalez, Low-power processor design,Technical Report No. CSL-TR-97-726, June 1997, Computer Systems Laboratory Departments of Electrical Engineering and Computer Science, Stanford University.

[3] R. Krishnamurthy, Mixed Swing Techniques for Low Energy/Operation Datapath Circuits, PhD thesis, Electrical and Computer Engineering, Carnegie Mellon University, December 1997.

[4] B. Joy (Ed), G. Steele, J. Gosling, G. Bracha, Java™ Language Specification (2nd Ed), Addison Wesley Pub., 2000.

[5] T. Lindholm and F.Yellin, The Java™ Virtual Machine Specification, Addison Wesley Pub., 1997.

[6] B. Venners, Inside the Java Virtual Machine, McGraw Hill, 1998.

[7] K. Cooper and N. McIntosh, Enhanced code compression for embedded RISC processors, Proc. ACM SIGPLAN '99 Conf. on Programming language design and implementation, May 1-4, 1999, Atlanta, GA, pp.139-149.

[8] H. Lekatsas, J. Henkel, W. Wolf, Code Compression for Low Power Embedded System Design, Proc. of the 37th Conf. on Design automation, June 5 - 9, 2000, Los Angeles, CA USA.

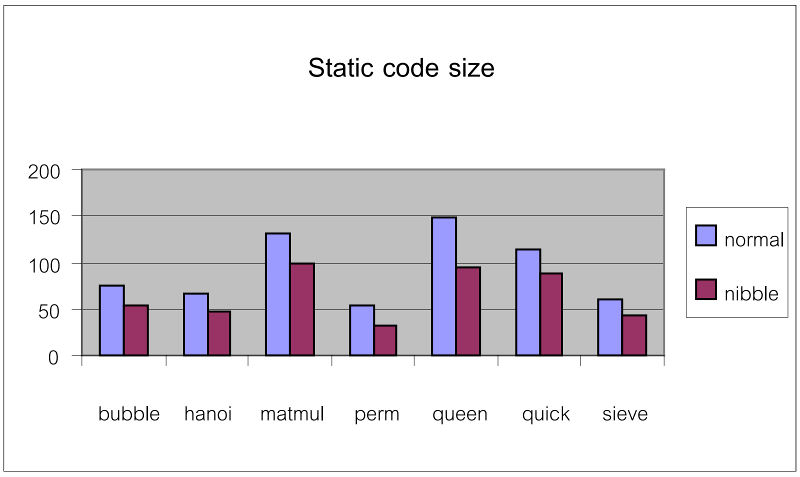
[9] P. Chongstitvatana, "Post processing optimization of byte-code instructions by extension of its virtual machine", 20th Electrical Engineering Conference, Thailand, 1997.

[10] J. Hennessy and P. Nye, "Stanford Integer Benchmarks", Stanford University.

[11] P. Chongstitvatana, "S2 processor and its opcode format", http://ww.cp.eng.chula.ac.th/ faculty/pjw/teaching/ads

	bubble	hanoi	matmul	perm	queen	quick	sieve
static							
normal	75	67	132	53	149	114	61
nibble	53	48	99	33	94	88	42
dynamic							
normal	11220	2153	2765	4539	384334	3595	2484
nibble	6668	1487	2058	2891	189719	2695	1385

Table 3: Statistics on the number of instructions on benchmark .



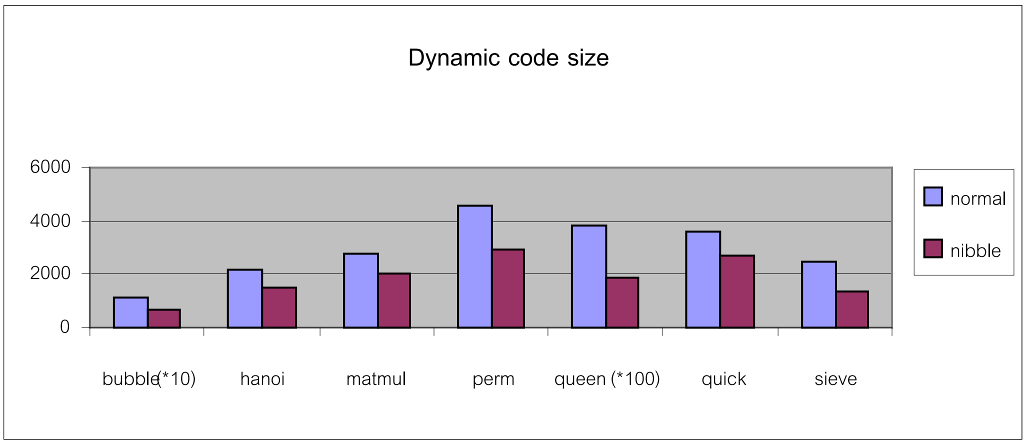


Figure 2: Static (top) and dynamic (bottom) code size for normal instruction set and nibble code instruction set.