Instruction compression by nibble coding: war on the old front.

Prabhas Chongstitvatana and Vishnu Kotrajaras

Department of computer engineering
Chulalongkorn University
Thailand
prabhas@chula.ac.th

Abstract

This work describes a new investigation of the problem of instruction set design: How to make the program as small as possible. The proposed method, called "nibble coding", compresses two instructions into one byte. The experiment is carried out to compare conventional byte-code instruction and a typical 32-bit machine code with the nibble coding. The result shows the proposed scheme achieves a smaller instruction bandwidth than a byte-code virtual machine and is much smaller than the conventional executable machine code.

Introduction

In the last decade the progress of microprocessor design has been phenomenal. Performance improvement rises according to Moore's law. The performance is double every 18 months. With this progress, there emerges strong leader of the manufacturers. The market competition forces the architecture to slowly converge. Instruction Set Architecture (ISA) has become not as important issue as the previous decade where the market was still young. Drive for performance is the key for the progress of the last decade. However, the 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 critical. This work describes a new investigation of the old problem: How to make the instruction (program) as small as possible. We experimented with instruction encoding 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 machine 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 architecture, 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].

Stack machines

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 oes not required addressing bit. Majority of instructions thus do not required 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] 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.

Code compression

There are a number of work in code compression [5, 6]. This work differs in that it is concentrated on the virtual machine code. The method to pack instructions into a smaller space in this work is based on the following techniques:

1. Extended instruction, making a special instruction that replace 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.
4. Packing two instructions into one instruction, this technique is called "nibble coding". This is the main contribution of this work.

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 [7]. 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. In this study we measured the use of this technique and found that it contributes only 10% of the code size reduction both in static measure and in dynamic measure. By static measure, we mean the size of the program. By dynamic measure we mean the counting of the number of byte of executed instructions, this measures the instruction bandwidth.

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 eliminate 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 for byte-code format the normal instructions occupies 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 maximised.
- 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 using a stack virtual machine. Various effects are measured to acertain their actual contribution to code compression. The detail of which will be describe in the next section.

Baseline machines

This section describes the stack virtual machine used in the experiment. The virtual machine is byte-coded and has the following instructions:

zero argument
arithmetic and logic

```
add, sub, mul, div, and, or, not,
eq, lt, gt
ldi, sti      ;;load/store indirect
ret           ;;return from call
```

one argument
varible access
```
get x, put x, ld y, st y  ;; access
to local (x) and global (y)
variables
```
control flow
```
jmp a, jtrue a, jfalse a, call
a   ;; a is address
```

and few other instructions to support machine execution and OS calls. An argument is 16-bit except local variable access which is 8-bit. A small benchmark suite to test integer instructions [8] is used to collect statistic of behaviour of code execution. The benchmark suite consists of seven programs (Table 1).

program aim description input

sieve test normal loop. Sieve prime number, the method of Erathothenes, find the prime <= 100
hanoi test recursion. Move 6 disks from peg 1 to peg 3.
matmul test loop and arithmetic. multiply 4 by 4 matrix, C = A * B
bubble test loop and swap. bubble sort, input data 20..1
qsort test loop and recursion. quick sort, input data 20..1
perm test recursion. Permutation generator. permute 4 numbers 0 1 2 3
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}

Table 1 the benchmark suite and its input. 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.

Experiment

Profiling the benchmark suite results in the following general observation:

- the top 10 most often used instruction consumes 88% of instruction bandwidth (Table 2).
- the most often used instruction is "get" (load local variables to stack) and its argument is 1..4
- the literal 0 and literal 1 constitute almost 100% of all literals executed.

get	30.5
ld	10.1
add	9.8
lit	8.1
call	7.8
ret	7.8
lt	3.6
jfalse	3.4
put	3.3
ldi	3.2

Table 2 the top 10 most often used instructions (percent)

Nibble coding

A code sequence can be compressed in many ways. However, various methods to compress the code interact with each other, for example, the expression `if x < 0` is translated into the sequence `get x, lit 0, lt, jfalse a1`. One can decide to use an extended code `jump-if-not-less-than-0` to replace the last three instructions, or the 2-combination `less-than-0` to compress `lit 0 + lt`, or the 2-combination `jump-if-not-less-than` to replace `lt + jfalse`. Choosing one of these will exclude the other choice. The nibble coding takes the highest priority in the design decision as we believe it will have the highest compression gain in overall.

The constraints in code compression are as follows:

- as the byte-code is used the natural boundary for accessing a code is a byte, the encoding of 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.

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 can not be the control flow: `jmp, jtrue, jfalse, call, ret`. Other 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 instructions will be considered separately.

Before deciding about the instruction selection for nibble coding the extended instructions are applied first as they maximally compress the size. The choice of incrementing and decrementing a local variable is obvious one as they are used in looping construct in most program (while, for).

```
get x, lit 1, add, put x  => inc x
get x, lit 1, sub, put x  => dec x
```

The instructions with arguments will be considered next to be combined to reduce their occurrence before nibble coding. The control flow instructions especially conditional jump are good candidates. The conditionals are: `eq, lt, gt` and the jumps are: `jtrue, jfalse`. These five instructions in combination cover all cases of conditional jump.

```
eq jtrue  => jeq     ;; jump if equal
eq jfalse => jne     ;; jump if not equal
lt jtrue  => jlt     ;; jump if less than
lt jfalse => jge     ;; jump if greater than
or equal
gt jtrue  => jgt     ;; jump if greater than
gt jfalse => jle     ;; jump if less than or
equal
```

Using the same reason of reducing the instruction with argument, literal 0 and literal 1 are considered for combining with other instructions. The top five 2-combination code with lit0/lit1 are (from most frequent) { `lit/add, lit/ret, lit/jmp, lit/sub, lit/eq` }. Their combination result in {`add1, ret0, ret1, 0jmp, 1jmp, sub1, eq0, eq1`} extended instructions. After applying the above extended instructions the legal lead instructions of the rest of the 2-combination code are found to be { `get, ld, add, ldi, put, sub, sti`}. From this set 8 intructions are chosen according to their frequency of use { `get1, get2, get3, get4, ld, add, ldi, sti` }. The follow set can have 16 instructions and the choice is the lead set plus { `ret, sub1, add1, sub, mul, st, call` }

(ordered by the frequency of use). The rule for combining the instruction is if the lead is "`ld`" then the follow must not be "`ld`" or "`st`" or "`call`" as only one argument is allowed in nibble coding. The argument is the index into the literal table. Some combination has zero argument.

Instruction encoding

The instructions are 8-bit. To maximise the size of index to the literal table the argument is 10-bit with 2-bit embed into the instruction. The normal and extended 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 (the `ld, st, call` instruction) then the nibble has one byte argument. (Fig 1)

normal and extended

```
0|op:7              ;; zero argument
0|op:7 a2:8         ;; local: get, put, inc, dec
0|op:5|a1:2  a2:8  ;; jmps, call, ld, st
```

nibble

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

Figure 1 Instruction encoding of the proposed scheme

A number of highly frequently used instructions are specialised to eliminate their arguments. They are {`get1, get2, get3, get4, put1, put2, put3, put4, lit0, lit1`}. Some of them will escape the code compression process and will use this final-catch compression.

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 using the calculation 1-A/B.

From Fig. 2, encoding arguments from 16-bit into 8-bit index into the literal table alone reduces the code size both static and dynamic in average 38% (e/base). Using extended instructions contributes 9% and 12% in static and dynamic code size reduction (ex/e). The

nibble coding affects 11% and 22% of further static and dynamic code size reduction (exn/ex). The combined effect of all methods is 50% and 57% of static and dynamic code size reduction (exn/base). In other words, using the proposed method the static code size is half of the base size and the dynamic code size is less than half of the base size. Comparing with a typical 32-bit 3-address machine code of a load/store register processor, S2 [9], the proposed method achieves the static code size of 74% less and the dynamic code size of 70% less than S2 (exn/S2).

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(TM) Language Specification (2nd Ed), Addison Wesley Pub., 2000.

[5] 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.

[6] 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.

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

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

[9] 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							
base	283	226	417	265	519	452	346
e	178	139	257	168	324	278	214
ex	162	133	230	154	296	246	196
exn	143	117	199	138	258	219	179
S2	596	400	892	444	1080	828	680
dynamic						x100	
base	40195	7203	6279	17428	32341	16096	12739
e	25493	4432	4031	10957	20376	9892	7829
ex	22861	4024	3628	9583	18321	8500	6703
exn	17189	2913	2705	7738	14954	6995	5344
S2	59444	14548	13752	24984	34216	19904	18288

Table 3 Statistics on the size of code (in bytes) on benchmark execution of several code compression methods. base = baseline byte-code encoding, e = compact encoding with literal table, ex = e + extended instructions, exn = ex + nibble coding, S2 = typical 32-bit 3-address machine code.

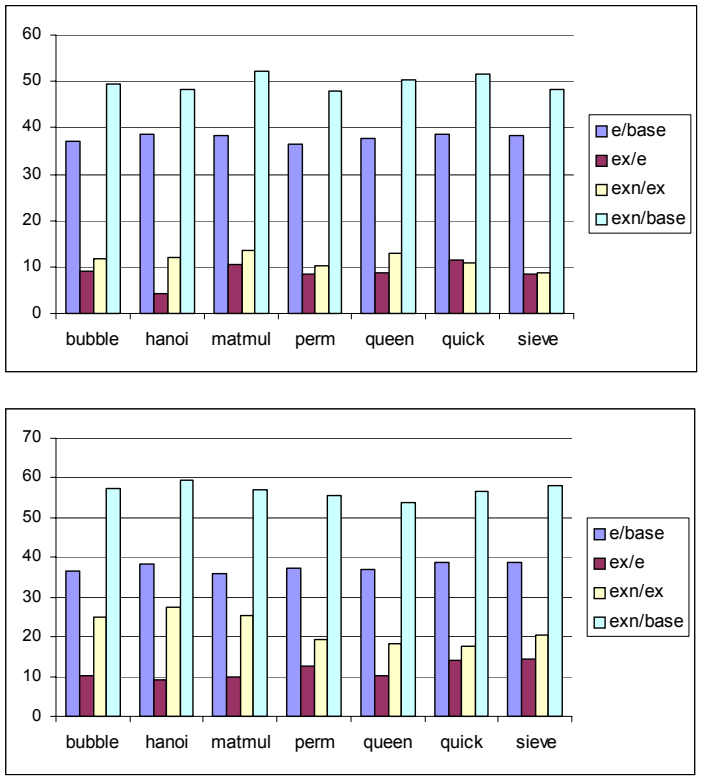

Figure 2 Percent of static (top) and dynamic code size reduction (bottom) of several code compression methods. base = baseline byte-code encoding, e = compact encoding with literal table, ex = e + extended instructions, exn = ex + nibble coding, S2 = typical 32-bit 3-address machine code.