Augmenting a Stack-based Virtual Machine with One-address Instructions for Performance Enhancement

Peera Thontirawong and Prabhas Chongstitvatana
Department of Computer Engineering
Chulalongkorn University, Bangkok, Thailand
u45pth@cp.eng.chula.ac.th, prabhas@chula.ac.th

***Abstract*— This work proposed a performance improvement of a stack-based virtual mchine by augmenting the instruction set with one-address instructions. The extended instructions are binary operators with two addressing modes: access local variables and immediate mode. An experiment is carried out to measure the effectiveness of the proposal based on modification of a stack-based virtual machine to include one-address instructions. A suite of benchmarks is used to measure both the number of instruction executed and the actual running time. The result shows that the proposed instructions reduce the number of instruction executed by 30% and the extended virtual machine is 29% faster than the original virtual machine.**

***Index Terms*— Stack-based virtual machine, one-address instruction, performance**

I. Introduction

Embedded systems for appliances are moving towards more complex systems. To keep the development time short and to ensure quality of software, a higher-level language for application development is preferred. The trend towards moving to Java platform is the case. The main advantage of Java is that the major source code of the application software is quite independent of the underlying hardware.

This is possible because in a Java platform the application software is compiled into a processor neutral executable code, Java byte-code. The execution of Java byte-code is done with a Java Virtual Machine [1]. This decoupling of application software with its execution environment via virtual machines promote portability. The slogan of Java is "write once run everywhere" reflects this concept.

Java Virtual Machine is a stack-based machine. Its instruction is based on zero-address instruction format. This makes the executable code very compact [2]. A stack-based machine is well-known to have a performance penalty. This is due to the bottleneck in accessing the central data structure, the evaluation stack. Every instruction accesses the stack. This is in constrasted to a register machine where multiple access to register bank is possible. Another performance limiting factor of stack-based instructions is that the number of instruction executed (the dynamic instruction count) is larger when compared to conventional three-address instructions. This also gives rise to a larger number of instruction to be fetched [3].

To improve the performance of a stack-based virtual machine, this work proposes the addition of one-address instructions to the virtual machine. Our experience with stack-based virtual machines shows that one of the most frequently used instruction is the access to a local variable, mostly pushing its value to the stack [4]. In zero-address instruction format, any operation takes its operand from the stack.

Therefore the operands must have been on the stack by some other operation, very frequently, the "get a local variable" operation. One-address instruction format can "compress" this sequence of instructions into one instruction.

To validate this idea, an experiment is performed based on an open source virtual machine S-code. S-code is a stack-based virtual machine publicly available [5]. A set of one-address instructions has been implemented as an extension to this virtual machine. The effect of this extension on the performance of the virtual machine is measured.

The paper is organised as follows. The next section introduces the S-code virtual machine. Section 3 explains one-address instruction extension. Section 4 presents the experiment and the results.

II. The S-code virtual machine

S-code virtual machine has a stack-based instruction set. S-code is designed for simplicity; the emphasis is on a small number of instructions. It is also quite fast to be interpreted by a software virtual machine. From S-code, it is easy to generate machine dependent code for a specific purpose, such as, small code size (byte-code, nibble-code) [4], high performance (extended code) [6], or to fit a particular hardware. There are also a number of real processors that use this instruction set, for example [7].

S-code has a fixed-length 32-bit instruction format. It is not compact but it is reasonably fast when interpreting. This format simplifies the code address calculation and allows code and data segment to be the same size (integer) as opposed to other format such as the byte-coded instruction format (as in JVM [1]). There are two types of instructions: zero-argument and one-argument. The zero-argument instructions are mostly

related to the arithmetic and logic operations. The one-argument instructions are the access operations to variables and the control-flow operations.

Each instruction is 32 bits. The right-most 8-bit is the operational code. The left-most 24-bit is an optional argument. For a virtual machine, this format allows simple opcode extraction by bitwise-and with a mask without shifting, but it needs 8-bit right-shift to extract an argument. Because zero-argument instructions are used more often, this format is fast for decoding an instruction.

Let's study some examples of programs in S-code (see Fig.1). Let a, b, c be locals; d, e be globals; L, M be labels. S-code is shown in Arial font.

```
a = a + 1
    get a, lit 1, add, put a

a = b[i]
    get b, get i, ldx, put a

d[i] = b
    ld d, get i, get b, stx

e = add2(a,b)
    get a, get b, call add2, st e

if (a == 1) then b = 2 else b = 3
          get a, lit 1, eq, jf L,
          lit 2, put b, jmp M,
    L:    lit 3, put b,
    M:
```

Fig. 1 An example of S-code

Here is a brief description of the instructions:
get x -- get a local variable and push it to the stack
ld/st a -- load/store a value from/to a global variable
lit c -- push a literal (constant) to the stack
ldx/stx -- load/store vector (array)
call f -- call a function
jmp/jf -- jump to a label, conditional jump
add/eq -- binary operators

III. One-address extended instructions

Naturally, the one-argument format is used for the extended instructions. There are two kinds of operands: local variables and literals. The S-code instruction set has 16 binary operators. These operators are extended to have additional two modes. The first mode has a local variable as the argument. Therefore, one operand of the operation is directly specified in the instruction, another operand resides on the stack. Similary, the second mode is the immediate mode which the argument is a literal. For example, the original zero-argument binary operators such as {add, sub, mul, div} have their companion instructions as {addv x, subv x, mulv x, divv x} for local variable mode and {addi c, subi c, muli c, divi c} for the immediate mode. These extended instructions effectively compress two instructions of getting a local variable to the stack and then operate on it into one instruction. Please observe this change from the previous examples (see Fig.2), now the executable code is shown with the extended instructions (in *Italics*). Let a, b, c be locals; d, e be globals; L, M be labels.

```
a = a + 1
    get a, addi 1, put a

a = b[i]
    get i, ldxv b, put a

d[i] = b
    get i, get b, stxv d

if (a == 1) then b = 2 else b = 3
          get a, eqi 1, jf L,
          lit 2, put b, jmp M,
    L:    lit 3, put b,
    M:
```

Fig. 2 The extended instruction of S-code (in Italics)

This extension adds 32 instructions to the original instruction set. The operation of these instruction is not complex. The extended instruction is just larger (almost double in the number of instruction).

IV. Experiment

In this section the effect of the additional one-address instructions in the extended virtual machine is measured by running a suite of benchmarks. There are seven small programs and two medium size programs used in the measurement: Bubble, Hanoi, Matmul, Perm, Queen, Quick, Sieve, AES and Compiler. The short description of the programs is as follows.

Bubble	Bubble sort 20 items of data. The initial data is ordered in descending order.
Hanoi	Solve the 6 disks Tower of Hanoi problem.
Matmul	Multiply 4 by 4 matrix using subroutine multiplication.
Perm	Generate permutation of 4 items
Queen	Find all solutions of 8-queen problem.
Quick	Quick sort 20 items of data (similar to Bubble and Merger).
Sieve	Find all prime numbers which are less than 500.
AES	AES (Advance Encryption Standard) (128, 128) bit key block cipher [8].
Compiler	Compile the compiler itself (Som language version 3.0 [9]). The compiler source is approximately 2500 lines of code.

The metric is the number of instruction executed to complete the tasks. As this is a measurement of a virtual machine, measuring its actual runtime does not yield much

useful result. The actual running time will depend too much on the implementation details of the virtual machine. The number of instruction executed is a fair measurement. However, although the number of instruction executed is reduced, other overhead (which is not related to the number of instruction executed) in the virtual machine may offset the gain. The actual running time is also measured using the same virtual machine (so that their implementations are as close as possible). This is achieved by running both instruction sets on the same virtual machine. The one-address instructions are in addition to the original instruction set therefore they are fully compatible. Table 1 shows the number of instruction executed for the extended virtual machine (zero+one address) versus the original (zero address). Table 2 shows their actual running time. The figures are calculated from the average of three runs.

Table 1. Comparing the number of instruction executed of two virtual machines.

Program	zero+one (1)	original (2)	(1)/(2)
Bubble	6594	10072	.65
Hanoi	1647	2310	.71
Matmul	10106	13626	.74
Perm	2826	4860	.58
Queen	244927	443324	.55
Quick	2406	3170	.76
Sieve	8446	13402	.63
Aes	20559	30684	.67
Compiler	5044736	6654829	.76
average			.67

Table 2 Comparing the actual running time (in ms) of two virtual machines.

Program	zero+one (1)	original (2)	(1)/(2)
Queen	728	1352	.54
Bubble	40	60	.67
Matmul	67	93	.71
Aes	140	187	.75
Compiler	41300	46900	.88
average			.71

The results show that in terms of the number of instruction executed, with the extended instructions the benchmarks are completed in 33% less than the original instruction set alone. In terms of the actual running time, the extended virtual machine is 29% faster than the original. This result is quite good in the sense that reduction of the number of instruction executed is translated into an improvement of the running time.

V. Discussion and Conclusion

The result clearly shows that extending the instruction set to include one-address format did improve the performance of a stack-based virtual machine by 30%. This enhancement is achieved quite readily, the modification of the virtual machine is not difficult as the new instruction format is compatible with the original one.

An intriquing idea is that if one-address format has such a good impact on performance what will happen if we consider two-address format? For example, in sorting benchmarks the sequence of swapping two elements in an array is executed very often. Suppose a few two-address format instructions are introduced for this purpose, such as load/store vector:

ldxa index base -- load base[index] to the stack
stxa index base -- store a value from the top of stack to base[index]

The two-address instruction format will need two arguments: the first argument, the local variable is 8 bits, the second argument the base address is 16 bits. With this simple addition, the following code is almost as good as possible (see Fig. 3).

```
swap(ar,i,j)    -- swap ar[i] and ar[j]
  t = ar[i]
  ar[i] = ar[j]
  ar[j] = t

      ldxa i ar, put t,
      ldxa j ar, stxa j ar,
      get t, stxa j ar
```

Fig. 3 Compiling a swap function into two-address instructions

Measuring this modification on sorting benchmark, the result is that the number of instruction is further reduced by 22%. Of coure, in terms of actual running time the two-address instructions are more complex hence will run slower per instruction than the zero-address instructions. But the overall gain is probably positive. However, not all situations are as good as this example. The only fact against ldxa/stxa is that the second argument, which is the address, is restricted to 16 bits. This is some how make it a special case, in general the address is 24 bits.

Two-address format creates a situation of explosion of number of instructions because two arguments have many combinations, such as ldxa above, the second argument can also be a local, a constant etc. Because of this, the instruction set will be very much incomplete and full of special cases. The tradeoff has to be made somewhere. Overall, we believe that including one-address instructions to a stack-based virtual machine has very clear benefit to the performance..

REFERENCES

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

[2] P. Nanthanavoot and P. Chongstitvatana, “Code-Size Reduction for Embedded Systems using Bytecode Translation Unit,” Conf. of Electrical/Electronics, Computer, Telecommunications, and Information Technology (ECTI), Thailand, 13-14 May 2004.

[3] S. Hines, G. Tyson and D. Whalley, “Reducing instruction fetch cost bypacking instructions into register windows,” Microarchitecture, 2005. MICRO-38. Proceedings. 38th Annual IEEE/ACM International Symposium on Microarchitecture MICRO 38, 12-16 Nov. 2005.

[4] N. Kotrajaras, and P. Chongstitvatana, “Nibbling Java byte code of resource-critical devices,” Proc. of National Computer Science and Engineering Conference, Thailand, 2003.

[5] http://www.cp.eng.chula.ac.th/~piak/project/som/s-code.htm

[6] P. Chongstitvatana, “Post processing optimization of byte-code instructions by extension of its virtual machine,” Conf. of Electrical Engineering, Bangkok, 1997.

[7] A. Burutarchanai, P. Nanthanavoot, C. Aporntewan, and P. Chongstitvatana, “A stack-based processor for resource efficient embedded systems,” Proc. of IEEE TENCON 2004, 21-24 November 2004, Thailand.

[8] J. Daemen and V. Rijmen, “The Rijndael Block Cipher: AES proposal,” 1999.

[9] http://www.cp.eng.chula.ac.th/~piak/project/som/index.htm