Performance comparison between two virtual machine interpreters : stack-based vs. register-based

Chatchawan Wongsiriprasert and Prabhas Chongstitvatana
Department of computer engineering
Chulalongkorn University
Phayathai road, Bangkok 10330, Thailand
phone (662) 218-6982, fax (662) 218-6955
email : chatchawan.w@chula.ac.th

Abstract

This work proposes a technique to speed up the execution of an interpreter for a virtual machine implementing in a high level language. We observe that for a stack-based machine the performance limit of the interpreter is likely to be the fetch-limit, i.e. the time spending on fetching and decoding an instruction. We hypothesise that to improve the performance the number of instruction to be executed should be reduced. This can be achieved by designing an instruction set that each instruction performs as much work as possible. Based on this assumption, we investigate an obvious alternative architecture -- a register-based machine. We have compared two virtual machines : stack-based and register-based. Using Stanford integer benchmark suite, the result shows that register-based virtual machine interpreter is 1.5 to 2 times faster than the stack-based virtual machine interpreter.

1. Introduction

The most popular applications based on virtual machine are Java applets. Its popularity derived from its ability to be platform independent. A Java compiler produces Java virtual machine instructions which are architectural neutral. This leads naturally to the style of implementation that uses an interpreter. There is a penalty of using interpreter, that is its performance. Many researches are attacking this performance issue [1,2]. These attempts are along these approaches : execution in hardware, just in time compiler and variants of compiler. Our previous work proposed a technique based on a virtual machine extension [3]. The optimization technique recognises the sequence of byte-code in the basic blocks. By analysing the dynamic execution of benchmark programs the most frequently used sequence are identified and replaced by the specialised version of byte-codes which are an extension of the original virtual machine.

An extension of virtual machine approach depends on the dynamic characteristics of applications, i.e. the frequency of the execution of particular sequence of instructions. From our earlier experience [3], the speed up is associated with the reduction of the number of stack operation. As a virtual machine is implemented in a high level language, we hypothesis that the main bottleneck is the time spending in the instruction fetch and decode, so called "fetch-limit". A basic loop in an interpreter is :

```
while(runflag) {
  execute byte-code at pc
  pc = pc + 1
}

execute(byte-code) {
  switch( byte-code)
    case ... :
    case ...
}
```

The number of times a program goes through the main `while` loop and `switch` dictates the fetch time. We explore an alternative Instruction Set Architecture (ISA) that aims to reduce the fetch time. By making each instruction do the work and much as possible the dynamic instruction count can be reduced, hence the reduction of fetch time. Towards this goal an obvious architecture to explore as opposed to the stack-based architecture (SVM) is the register-based architecture (RVM). In the register-based architecture an instruction has access to a number of operands in the registers instead of limit to the access to the stack. The registers can be accessed randomly unlike the stack. Another possible alternative is the Long Instruction Word (LIW) architecture. The LIW approach requires a compiler that can perform instruction scheduling. As an interpreter is basically sequential in execution, a long instruction needs to be unpacked and execute each short instruction individually which in the end the execution time will be comparable to a register machine. The other alternative is the vector architecture where an instruction can perform a vector operation. This will definitely reduce fetch time but a vector machine is hardly a general purpose architecture. It is limited to applications that are suitable for vectorization.

The next section describes how we design the experiment and how we test the hypothesis. The subsequent sections explain the stack-based machine and register-based machine in details and discuss the experiment and the result.

2. Comparing the SVM and the RVM

The test bed is a set of small integer benchmark (Stanford integer suite) [4]. The SVM comes from [3]. It is a simple stack machine. The RVM is basically a 3-operand register machine. We compare the dynamic instruction count and the execution time. Comparing the instruction count doesn't pose any problem as it depends mostly on the instruction set design although the compiler affects the quality of the generated code, we try to generate a reasonable good code for both machines. Comparing the execution time depends on the implementation of the virtual machine interpreter. In this respect, it is difficult to separate the difference caused by the implementation and the difference caused by the instruction set design. We try to minimise the difference of the implementation by using the similar style in implementing both virtual machine interpreters. We believe that the reported statistics show the difference caused by the instruction set design fairly well.

2.1 The register-based machine (RVM)

The RVM is a 3-operand register machine. It is a load-store architecture with 32 registers. The machine has a 2^{32} words addressable unified memory space (figure 1-2). The operand in the opcode can be accessed using 5 addressing modes: (1) absolute (with an address size of 26 bits), (2) intermediate (-2^{19} to $2^{19}-1$), (3) base/index, (4) base/displacement (with an address offset size of 15 bits) and (5) register deferred. The machine has 29 32-bit general-purpose registers (GPRs), namely R2, R3 … R30. The R0 is reserved for a constant zero. R0 is used in the same way as GPRs but its value always 0 when read and the write to R0 is ignored. R1 is a program counter and R31 stores the remainder in the DIV instruction.

The RVM has 15 simple instructions. Later, after some experiments, we found that we can speed up the execution by adding 2 stack operations (SAVE and RSTO) to save and restore value of registers between a function call. All of the instructions support only 32-bit word data type. The instructions are encoded using 32-bit fixed length encoding. The instructions can be divided into three groups, (1) data transfer: - LOAD, STORE, LOADI, SAVE and RSTO. All instructions in this group except LOADI are the only instructions that can access the memory. (2) Control flow: - CALL, JUMP and HALT and (3) ALU instructions: - ADD, SUB, AND, OR, XOR, SHIFT, MUL and DIV. The ALU instructions have an option to set conditional flags (zero, negative, carry and overflow) to reflect the result of the instruction.

31	26	25	24	19	14	9
OP	0	F	Result (RC)	First (RA)	Second (RB)	(unused)

31	26	25	24	19	14
OP	1	F	Result (RC)	First (RA)	Unsigned literal ($0 - 2^{15}$)

F : 1 Set conditional flags

Figure 1. ALU register and literal format

31	26	24	19
OP	00	Dest (DS)	Absolute address ($0-2^{20}$)

31	26	24	19	14
OP	01	Dest (DS)	Base (BS)	Relative Displacement ($-2^{14}-2^{14}$)

31	26	24	19	14	9
OP	10	Dest (DS)	Base (BS)	Index (IDX)	(unused)

Figure 2. Data transfer : absolute , base displacement and base index format

2.2 The stack-based machine (SVM)

The SVM is a byte-code machine. The byte-code set is quite minimal. It has been designed to make it easy to implement the interpreter for various platforms. Figure 3 shows the semantics of byte-codes.

Notation : M memory, DS data segment, SS stack segment. Aop arithmetic operators, Lop logical operators, Uop unary operators.

Byte-code	Operational semantics
[Lit #n]	push(n)
[Lvalg #ref] /1/	push(ref)
[Lval #i] /1/	push(Fp-i)
[Rvalg #ref] /1/	push(DS[ref])
[Rval #i] /1/	push(SS[Fp-i])
[Fetch]	push(M[pop])
[Set]	M[pop1] = pop2
[Index] /2/	push(base_ads + index)
[Jmp #ads]	Ip = ads
[Jz #ads] /3/	if pop = 0 then Ip = ads
[Call #ads] /4/	push(Ip), Ip = ads
[Func #np #nl] /5/	save state, new stack frame, pass parameters
[Ret0]	remove stack frame, restore state
[Ret1]	remove stack frame, restore state, return a value
[Stop]	terminate the process
[Aop]	push (pop1 Aop pop2)
[Lop]	push (pop1 Lop pop2)
[Uop]	push (Uop pop)

Figure 3 SVM byte-code semantics

/1/ variable access
/2/ effective address calculation for array var.
/3/ if top of stack = 0 jump
/4/ call to subroutine
/5/ create new stack frame, invoke a function

3. Result and discussion

The percent of reduction of the instruction count is shown in Figure 4.

$$percent\ reduction = (I_{svm} - I_{rvm}) * 100 / I_{svm} \quad (1)$$

where I is a dynamic instruction count. The instruction fetch is reduced 35-60%. This shows a big advantage of RVM over SVM. This advantage is translated to the reduction of the execution time. Figure 5 shows the speed up [5] of RVM over SVM.

$$speed\ up = T_{svm} / T_{rvm} \quad (2)$$

where *T* is the execution time. The speed up is proportional to the reduction in the instruction count, 1.5–2.0. Although the fetch time in RVM in reduced, the execution of each instruction is slightly increased because each instruction does more work than each instruction in SVM. Decoding of an instruction in RVM is also more complicate due to its addressing mode. The programs that heavily use recursive call do not gain much speed up because the cost of function call is similar between RVM and SVM. Finally the size of executable code is compared in figure 6. The size of executable code of both ISA are comparable. Each instruction of RVM is larger (32-bit) but there are fewer instructions in a program.

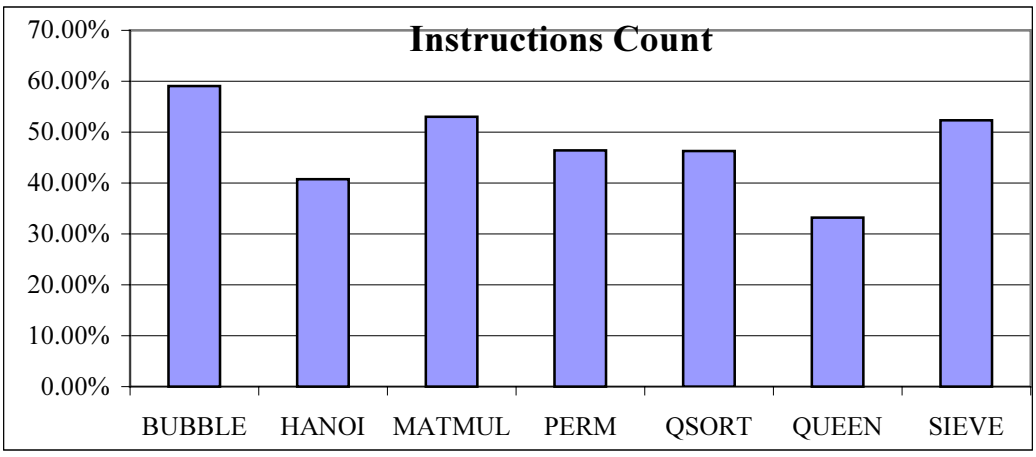

Figure 4 The reduction of the number of instruction count of RVM over SVM (%)

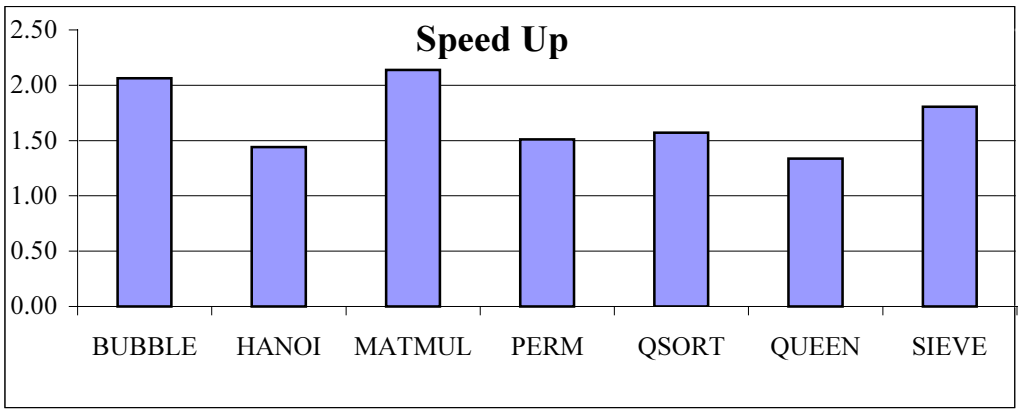

Figure 5 The speed up of RVM over SVM

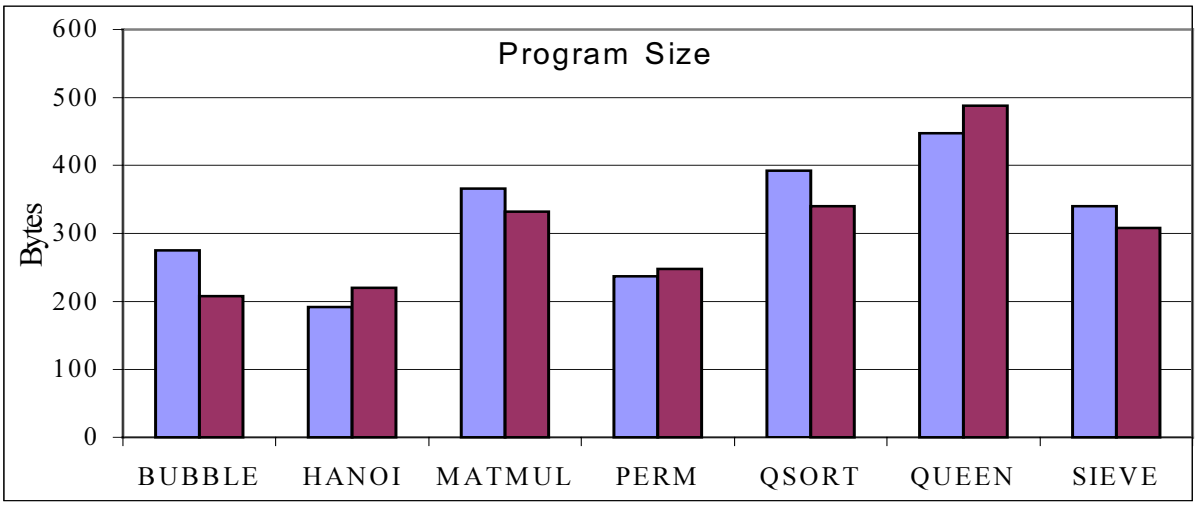

Figure 6 Comparison of the executable size

4. Conclusion

From the analysis of execution profile of SVM we hypothesise that the performance is fetch-limit. We propose a RVM which is designed to minimise the dynamic instruction count. The ISA of SVM and RVM are compared using Stanford integer suite benchmark. The result of the experiment shows that RVM has reduced the instruction count 35-60% compared to SVM. This advantage carries to the execution time speed up of RVM over SVM. This result confirms our hypothesis.

In the future, we believe that the implementation of RVM can be improved. Our future work will concentrate on the direct translation from byte-codes to the instruction set of RVM without recompilation of the source code. This can be applied to improve the execution speed of byte-code execution in JavaVM by introducing a suitable RVM for Java. The advantage of this approach is that the platform independence of Java is retained while the execution speed is improved. The scheme is also compatible with the existing structure of Java application deployment.

References

[1] T. Cramer and others, “Compiling Java Just In Time”, IEEE Micro, Vol.17 No. 3, 1997.
[2] H. McGhan and M. O’Conner, “PicoJava : a direct execution engine for Java bytecode”, IEEE Computer, Vol.31 No. 10, 1998.
[3] P. Chongstitvatana, "Post processing optimization of byte-code instructions by extension of its virtual machine", Proc. of 20th Symposium of Electrical Engineering, Bangkok, Thailand, 1997.
[4] J. Hennessy and P. Nye, "Stanford Integer Benchmarks", Stanford University.
[5] J. Hennessy and D. Patterson, “Computer architecture : a quantitative approach”, 2nd edition, Morgan Kaufmann Pub., 1996.