Proc. of the 20th Conf. of Electrical Engineering, Bangkok, 1997

Post processing optimization of byte-code instructions by extension of its virtual machine.

Prabhas Chongstitvatana
Department of Computer Engineering, Chulalongkorn University,
Phaya Thai Road, Bangkok 10330, Thailand.
Phone (66-2) 218-3721, Fax (66-2) 215-3554, E-mail : fengpjs@chulkn.car.chula.ac.th

Abstract

This work describes an optimization technique which is applied to post-process byte-code instructions. By extending a virtual machine of an instruction set, some sequence of byte-codes can be represented by a shorter code which can be executed faster. The experimental result showed that this technique yielded 25% - 120% speedup on Stanford integer benchmark suite.

1. Motivation

This work describes an optimization technique which is applied to post-process byte-code instructions. By extending a virtual machine of an instruction set, some sequence of byte-codes can be represented by a shorter code which can be executed faster. It is significant that this is a post-processing method, it is aimed to be applied to the already compiled program. There is no access to the source program. There are various uses of this technique, for example, to speed up the execution of JAVA applet [8,9] (which is byte-coded) that has been loaded into a browser, one has no access to the source program in such case.

Motivation of this work came from an observation that a virtual machine for byte-code instructions aimed for portability and architectural neutral mostly concerns orthogonality of the instruction set. Therefore it is possible to specialise some sequence of codes to gain speed.

Optimized compilation of conventional languages to stack-based virtual machine has received relatively little attention. Previous work has focused on using the stack efficiently for expression evaluation [1,2] and performing peephole optimizations [3,4]. Koopman [6] has done a work on optimizing stack usage at the basic block level and the global level. He performed intra-block stack scheduling which reported 91% - 100% elimination of redundant local variable accesses.

This work differs significantly from previous work in the premise that the technique reported is applied to byte-code, not at the source code level. In this regard, this work can be described as a bottom-up approach for code optimization as opposed to top-down approach in the optimization is applied to source code compilation.

2. Methodology

Main idea for our bottom-up approach is to reconstruct basic blocks from the byte-codes which are compiled from benchmark programs (static analysis). We perform the analysis of the execution of these byte-codes and recognise the most frequently used sequences (grouping as basic blocks). From this dynamic analysis, the virtual machine is extended with specialised byte-codes to perform the most frequently used sequences. We select R1 concurrent system [7] as our base system for experimentation. R1 system composed of a compiler which compile R1 source language into byte-codes which will be executed by R1 run-time interpreter. The reason for choosing this system is simply that it is our own work therefore we have access to all details for its implementation. (A brief

description of R1 system is given in the next section).

We compiled seven integer benchmark programs from the Stanford Benchmark Suite [5]. These programs are : bubble sort, tower of Hanoi, matrix multiplication, permutation, quick sort, 8 queens problem and generate prime numbers by sieve method. Although significantly larger programs must be used to make conclusive statements about performance, these programs are sufficiently complex and varied at the basic block level to illustrate the effectiveness of our method.

3. R1 system

R1 is a simple language which provides concurrency control, protection of shared resources, interprocess communication and real-time facilities. The aim of this language is for it to be a small, simple, and practical language for programming an embedded application. The syntax of R1 is intentionally made to be "like" C language. So that the user who is familiar with C language can read and write R1 easily.

statements	examples
assignment	a = b + 1 ;
if-statement	**if** (expr) stmt-true [**else** stmt-false] ;
while-statement	**while** (expr) stmt ;
return-statement	**return** (expr) ;
function-call	function-name (actual parameters) ;
process-call	process-name (actual parameters) ;

Figure 1 Syntax of R1 language

A program is composed of declarations and a main. There are four types of declaration : global variable declaration, semaphore declaration, function declaration, and process declaration. There are several types of statements : assignment statement, flow-control if and while, and some additional function for concurrent processing. Operators are basic operators such as + - * / etc. including addressing operators '*' (dereference) and '&' (address). The scalar data is basically a word (which can be 16 bits or 32 bits depended on the architecture of the target hardware) with no type. Only one type of structured data is available, it is a one dimensional array of words. Global variables must be declared. Local variables are automatically declared and they are lexical-scoped. Local variables appear in the formal parameter list and can not be an array variable.

3.1 Operators

`( )`	do an expression inside ( ) first
`- ! * &`	unary op - minus, ! logic not, * deref, & address
`* / &&`	multiply, divide, logical and
`+ - ||`	add, subtract, logical or
`< <= == != >= >`	relational operators

Figure 2 Operators in R1 language

The type of operation is : word $\times$ word $\rightarrow$ word. Therefore the overflow and underflow can occur. All arithmetic operators treat a value as a signed integer.

3.2 Example of a R1 program

```
sort()
{
   i = 10;
   while(i) {
     j = 1;
     while(j < i) {
       if ( data[j] < data[j+1])
            swap(j,j+1);
       j = j+1;
     }
     i = i-1;
   }
}
```

For the purpose of this paper we run all programs as a single process and disable process switching. From now on we will ignore all the discussion concerning concurring processing and real-time aspect of the system.

3.3 R1 byte-code

The byte-code set of R1 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 which we will refer to in the rest of the paper.

Notation : CS code segment, 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
[Proc #pid #np #nl]	new process descriptor, initialise state, awake
[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 R1 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

4. Reconstruction of basic blocks

To identify basic blocks, we looked for byte-codes that can be used as "stop-word" such as the transfer of control : Jmp, Jz, Call, Ret. For example :

```
while ( i < n ) { body } =>
$1 [ rval i, rval n, LE ] Jz $2, [
body ] Jmp $1, $2
```

We can identify the basic block [rval i, rval n, LE] and [body] from the byte-code Jz and Jmp. To identify an individual statement, the "stop-word" are : Fetch, Set, Index. For example :

```
a = b + c; => lval a, rval b, rval
c, plus, Set.
b = c[];   => lval b, lvalg c, lit
2, index, Fetch, Set.
```

We tagged these basic blocks and statements and counted the frequency of their use during the run. After the analysis of the execution of byte-codes of the benchmark programs, we identified the most frequently used sequences as shown in Table 1.

Table 1 The most frequently used sequences

byte-code sequence	correspond to
lval a, rval a, lit 1, plus, set.	a = a + 1;
lval b, lvalg c, ... , index, ...	b = c[..] ...
lvalg c, ... , index, ...	c[..] = ...
lval a, lit 0, set	a = 0;
lvalg c, ..., index, lit 0, set	c[..] = 0;
lval a, rval a, exp, plus, set	a = a + exp;
lvalg c,..,index,lvalg c,..,index, fetch,..,plus set	c[n] = c[n] + ...
rval a, rval b, EQ, Jz	if (a == b)
rval a, lit 0, EQ, Jz	if (a == 0)
lvalg c, ..., index, rval b, LE, Jz	if (c[..] <= b)
rval a, rval b, LT, Jz	while (a < b)

We classified these sequences into 4 classes :

1. increment, decrement and combined operators (such as "+=" in C language).
2. array access
3. assignment
4. flow control

We defined an extension of R1 virtual machine to represent these sequences by special byte-codes :

Table 2 The extended byte-code

extended byte-code	for the sequence
inc v (dec v)	lval v, rval v, lit 1, plus, set.
addset a	lval a, rval a, exp, plus, set.
set-var a	lval a, ... set.
set-0 a	lval a, lit 0, set
EQjz a b $1	rval a, rval b, EQ, jz $1
Jnz a $1	rval a, lit 0, EQ, jz $1
LEjz a b $1	rval a, rval b, LE, jz $1
LTjz a b $1	rval a, rval b, LT, jz $1

We notice that some combination of variable accesses are more frequent than other so we specialised "addressing mode" of extended byte-codes further. In set , a variable can be either local or global therefore we defined :

```
set-local v,  set-global v.
```

The combined operator "+=" of an array variable :

```
c[m] = c[m] + exp => c[m] += exp
```

correspond to :

```
lvalg c, rval m, index, lvalg c,
rval m, index Fetch plus set. =>
exp, rval m, addset2 c
```

Also various form of "while a < b" where a, b can be : local, global, array. We selected two forms :

```
LTjz2 local local $1
LTjz3 local global $1
```

and one form of :

```
LEjz2 array local $1
```

for the statement if(c[..] < b) . Totally there are 21 additional instructions for the extended virtual machine.

5. Experimental results

The original byte-code programs were transformed using the extended byte-codes with all the offset of the Jmps and Calls readjusted properly. We analysed the execution of the transformed programs to observe the effect of each category of the optimization method :

1 apply only extended array access byte-code
2 apply only combined operators (increment, decrement, addset) byte-code
3 apply only extended assignment byte-code
4 apply only extended flow control byte-code
5 apply all the above methods

We reported the results of the speedup, the reduction in the number of stack operations and the reduction in size of the byte-code programs. The result of using all extended byte-code is that the speedup varies from 25% (hanoi) to 120% (sieve). The reason for low speedup in hanoi is because the hanoi program mainly is recursion which is not affected by our optimization. Combined operators byte-code is the most effective method which yields the result 20% - 65% speedup. This is not unexpected because all the loops contain increment or decrement of the loop counter which can be most effectively optimized by `inc v, dec v` extended byte-code. Part of the speedup is contributed by the reduction in the number of stack operations (push and pop). The aggregate result is 20% - 80% (Fig. 5). It is worth noting that the "shape" of the graph of Fig. 5 compared to the graph of Fig. 4 supports this reasoning. In terms of the reduction in size of byte-code programs, the aggregate result is 10% -34%.

6. Discussion

The bottom-up approach to optimization by using only post-processing of byte-codes has yield the result of 25% - 120% speedup. The optimization technique recognises the sequence of byte-code in the basic blocks and statements. 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. The application of this technique to the popular byte-code system Java [8,9] might proved to be interesting. Java aims to be architectural neutral, platform independent and safe. Its byte-code has been designed to be suitable for stack architecture hardware [10]. As opposed to another optimization technique, Just In Time compilation, which is platform dependent, the extended virtual machine is to a large degree platform independent. The extended byte-code can be regarded as the "specialization" of the original virtual machine for the type and mode related operations such as the access to local/global variable, the access to scalar/array operand, the combined operations such as increment/decrement and some branching operations.

7. Acknowlegdement

This work is funded by the faculty of engineering research center, Chulalongkorn university.

References

[1] J. Bruno and T. Lassagne, "The generation of optimal code for stack machines", JACM, July 1975, 22(3):382-396.

[2] J. Couch and T. Hamm, "Semantic structures for efficient code generation on a stack machine", Computer, May 1977, 10(5):42-48.
[3] J. Hayes, "An interpreter and object code optimizer for a 32 bit Forth chip", in 1986 FORML Conf. Proc. pp.211-221.
[4] T. Hand, "Performance of the Harris RTX-2000 C compiler", in Proc. of the 1989 Rochester Forth Conf., pp.61-62.
[5] J. Hennessy and P. Nye, "Stanford Integer Benchmarks", Stanford University.
[6] P. Koopman, "A preliminary exploration of optimized stack code generation", in Proc. of 1992 Rochester Forth Conf.
[7] P. Chongstitvatana, "A multi-tasking environment for real-time control", Progress report, Faculty of engineering research center, Chulalongkorn university. Also http://www.cp.eng.chula.ac.th/ faculty/pjw/ ISL.htm
[8] G. Tribble, "Java computing whitepapers", 1996, http: //www.sun.com/ javacomputing/ whpaper/
[9] J. Gosling and H. McGiltion, "Java language environment whitepaper", 1996, http://java.sun.com/doc/language-environment.
[10] Picojava, 1996, http://java.sun.com.

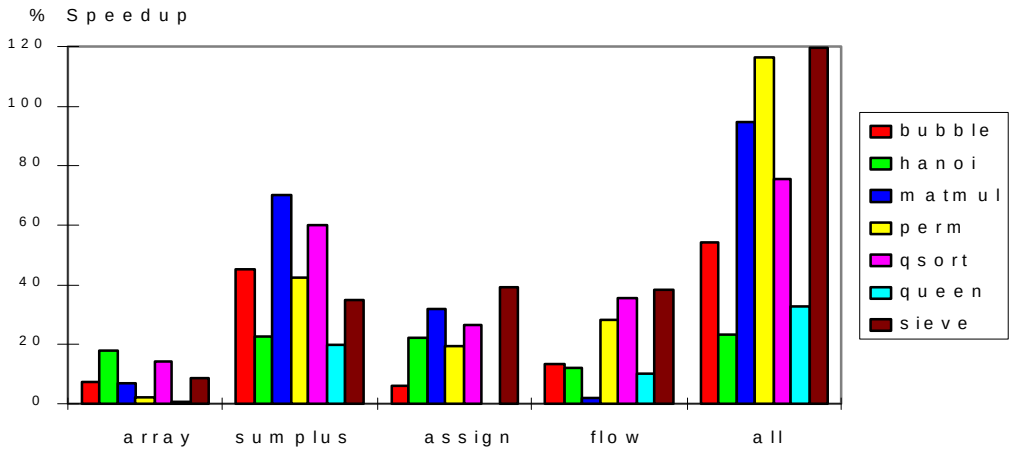

Figure 4 Execution speedup

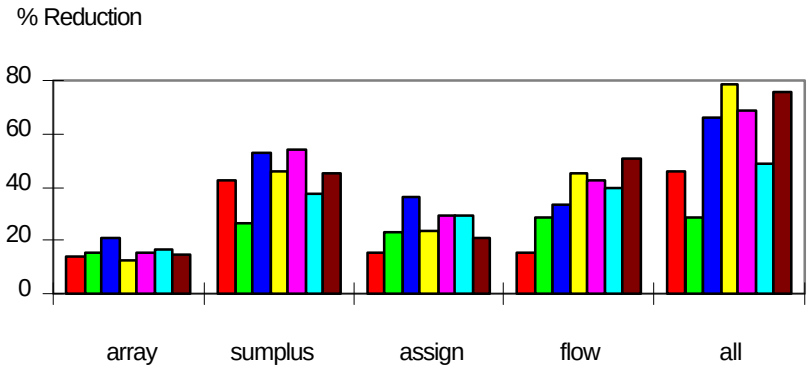

Figure 5 Reduction of the number of stack operations

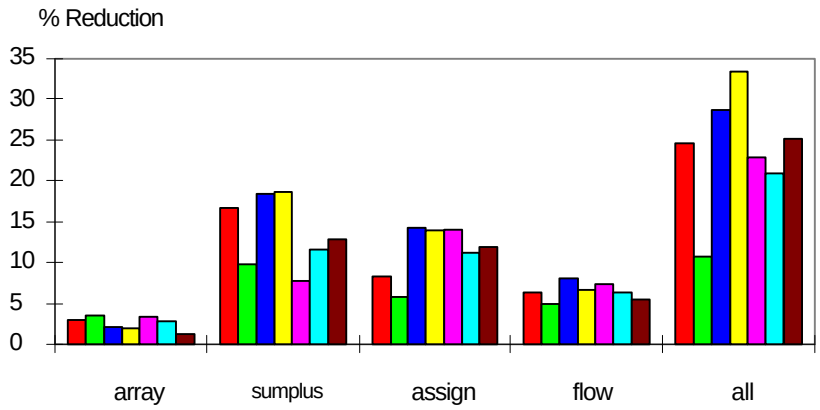

Figure 6 Reduction of the size of byte-code programs