Available online at www.sciencedirect.com
SCIENCE DIRECT®
Science of Computer Programming 57 (2005) 295–317
Science of Computer Programming
www.elsevier.com/locate/scico

Generation of fast interpreters for Huffman compressed bytecode

Mario Latendresse[a,*], Marc Feeley[b]

[a] *Science and Technology Advancement Team, FNMOC/U.S. Navy, 7 Grace Hopper, Monterey, CA, USA*
[b] *Département d'informatique et recherche opérationnelle, Université de Montréal, C.P. 6128, succ. centre-ville, Montréal, H3C 3J7, Canada*

Received 8 December 2003; received in revised form 15 June 2004; accepted 27 July 2004
Available online 12 May 2005

Abstract

Embedded systems often have severe memory constraints requiring careful encoding of programs. For example, smart cards have on the order of 1K of RAM, 16K of non-volatile memory, and 24K of ROM. A virtual machine can be an effective approach to obtain compact programs but instructions are commonly encoded using one byte for the opcode and multiple bytes for the operands, which can be wasteful and thus limit the size of programs runnable on embedded systems. Our approach uses canonical Huffman codes to generate compact opcodes with custom-sized operand fields and with a virtual machine that directly executes this compact code. We present techniques to automatically generate the new instruction formats and the decoder. In effect, this automatically creates both an instruction set for a customized virtual machine and an implementation of that machine. We demonstrate that, **without** prior decompression, fast decoding of these virtual compressed instructions is feasible. Through experiments on Scheme and Java, we demonstrate the speed of these decoders. Java benchmarks show an average execution slowdown of 9%. The reductions in size highly depend on the original bytecode and the training samples, but typically vary from 40% to 60%.

* Corresponding author.
E-mail addresses: mario.latendresse.ctr.ca@metnet.navy.mil (M. Latendresse), feeley@IRO.UMontreal.CA (M. Feeley).

0167-6423/$ - see front matter © 2005 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2004.07.005

1. Introduction

Embedded systems are resource-constrained devices requiring careful attention to memory usage and power consumption. To attain these goals, several researchers are taking the approach of reducing program size [6,15,14]. We focus on the context where code decompression cannot be performed prior to the program's execution. This constraint is reasonable for embedded systems where a bulk decompression of programs, or even parts of programs, before execution, might exceed the available RAM.

Embedded systems typically contain both RAM and ROM memory and in most systems ROM is much larger than RAM. If RAM is used to store program code, for example by downloading software from a server, and the virtual machine is stored in ROM, it is advantageous to use all the ROM space if it reduces RAM usage for programs. As demonstrated in this paper and by some other researchers [24], this may even increase execution speed. Therefore, in some cases, the compression factor[1] should be measured only for the bytecode stored in RAM with the constraint that the virtual machine fits in ROM; the objective is not to create a small virtual machine but rather to increase its size – as much as the ROM allows – in order to reduce the size of the program stored in RAM and/or increase execution speed.

In other situations, the size of the bytecode and the virtual machine must be taken into account: for example, when they are both placed in ROM, and the RAM is solely used for dynamic data. In such cases, the compression factor should be based on the bytecode and virtual machine sizes.

Some researchers [12,24,9] have shown the virtues of reducing code size, without decompression before execution, by using bytecode interpreters tailored for one program. Compression of the bytecode, capable of direct execution without decompression, would further reduce code size. Other researchers [10,12,3] have stated the possibility of using Huffman codes to compress bytecode, usually to conclude that if the decoding is done in software it increases execution time to an unacceptable level. In [10], it is further argued that the faster technique of look-up tables, using k bits, as presented in [4], uses a significant amount of space. Our solution is to use canonical Huffman code [27] and several smaller look-up tables to keep the total size small. Reducing space taken by operands is also important since they usually account for a large part of the code size. Instruction formats with small operand fields can further reduce size. The main goal of this paper is to show that there are techniques to efficiently decode such compressed instructions.

Typically, virtual instructions are "byte encoded": operational codes (opcodes for short) and operands are encoded in byte units. This method trades space for speed by maintaining byte, or even word, alignment and a fixed length for all opcodes. In this work, we use a variable bit-length encoding for a more compact form. We show that using canonical Huffman code for opcodes, new customized instruction formats, replacement of sequences of repetitive instructions by one opcode and no byte boundary alignment can significantly reduce bytecode size and still allow fast direct execution by an interpreter. For speed, canonical Huffman codes should not be decoded bit by bit; instead, blocks of k bits should

[1] The compression factor is the length (size) of the compressed code divided by the length of the uncompressed code (original bytecode). Therefore, a small factor means a good compression.

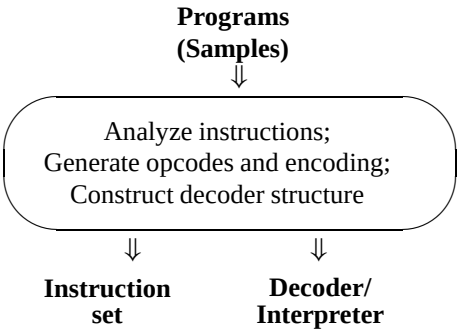


Fig. 1. Creation of instruction set, its decoder and interpreter.

be used. Such an idea has been explored previously by Turpin and Moffat [21]. We have extended their work to allow multiple k bit look-ups and generate decoders given a space constraint.

Our approach assumes automation. The decoders, the interpreters, and compressed instruction set, are all automatically generated (in C) by a tool. They are generated by specifying space constraints and providing a corpus of sample bytecode programs. Therefore, in our approach, the design of virtual machines and instruction sets for compressed programs can be automated. This context allows two application areas for the work presented in this paper. The first one is the design of virtual machines, such as the KVM for Java, aimed at embedded systems with memory constraints. This could be done for any language. The construction of such machines should be done based on a careful analysis of program samples. The second application is the compilation of programs where code space is a major concern. In that case, a virtual machine tailored for the program can be used to reduce space. This is the approach taken in [12,24,9]. Further code size reduction can be obtained with a compression of the virtual instructions.

In the next section, we give a general presentation of the compression algorithm. In Section 3 canonical Huffman codes are presented along with a compact but slow decoding method. Section 4 presents much faster but slightly less compact decoders. Section 5 explains the C code's structure for all canonical decoders. Section 6 discusses how decoders access memory for opcodes and operands. Experimental results showing that the approach is practical are presented in Section 7. Section 8 presents some of the related work.

2. The compression algorithm

Fig. 1 presents our general framework. The sample programs are bytecode encoded with an unmodified compiler. The samples should be appropriately chosen to represent the code that will be deployed. In particular, the same compiler should be used to create the samples and the deployed code. An instruction set encoding to compactly represent the samples is then generated by a tool. This requires an analysis of the instruction frequencies, the

length of operands, etc., in the samples. The decoder is generated given a space constraint parameter, along with the interpreter. The sizes of the decoder and interpreter are taken into account to reduce program sizes. This approach is transparent for the compiler writer since the compression of programs can be done from the original bytecode.

Our compression approach creates new instructions and encodings for them. These instructions are either macro-instructions that do the work of a sequence of other instructions, or basic instructions with a new format for the operands. We proceed as follows to create them. From the program samples:

(1) A dictionary of (possibly overlapping) repetitive sequences is built. We limit the size of this dictionary by limiting the length of the sequences and imposing a lower bound on the frequency of occurrence in the samples.
(2) A dictionary of formats to encode all basic instructions using as few bits as possible is created. It includes the original formats of the virtual machine in order to guarantee that all possible programs can be encoded.
(3) A greedy algorithm repetitively selects either a new format or a sequence of instructions, based on the maximum space saving, until no space gain can be obtained.

The greedy algorithm takes into account the opcode lengths, the new formats, and the space of the decoder. Further details on the selection algorithm can be found in [17].

Henceforth, the following setting is used: the opcodes are variable length canonical Huffman codes generated using the static frequencies of the opcodes from sample programs; and operands are uncompressed but of a length that is not restricted to a multiple of eight bits. Thus, opcodes and operands are not byte-aligned. The branch offset of branch instructions is measured in bits, but instructions following subroutine calls are byte-aligned—return addresses are in bytes.

3. Canonical Huffman encoding of opcodes

We encode opcodes using canonical Huffman codes [27]. These are similar to Huffman codes built by the original bottom up method of [13], but the numerical values of the codes of a given length form a consecutive sequence. As will be shown, they have a very compact representation of the bijection between the codes and the encoded object.

Huffman codes are typically generated by incrementally building a binary tree. In the case of canonical Huffman codes, the resulting tree has all its branches “pushed” to the right (or left). Fig. 2 shows some canonical codes mapped into a tree. The codes, when read left to right at the leaves, are in order of non-decreasing lengths and in order of increasing values. Moreover all the codes of a given length form a non-interrupted consecutive sequence of binary values (for example, all the codes of length four are `1100`, `1101`, `1110` and `1111`, i.e. 12, 13, 14 and 15).

Since we use canonical Huffman codes for opcodes, the two terms will be used interchangeably; and *canonical* will often be dropped as we only use canonical Huffman codes.

In the rest of this section we explain a compact representation of opcodes and an efficient flexible decoding technique for them.

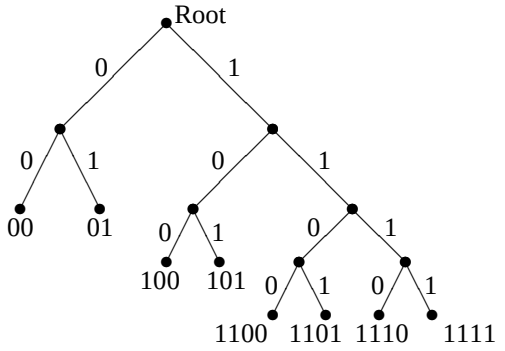


Fig. 2. A canonical ascending Huffman tree.

Let l_c be the length in bits of code c (a canonical Huffman code), $v(c)$ its value, $k \geq l_c$ a constant, and $V^k(c) = v(c)2^{k-l_c}$; in other words, $V^k(c)$ is the value of c left justified in a k bit processor register. Left justification allows the creation of a very compact decoder as presented in Section 3.1.

Let $C = \{c_i\}$ be a set of opcodes and $l_{\max}$ their maximum length. Assume that the opcodes are decoded in a variable (e.g. processor register) of w bits such that $w \geq l_{\max}$. Define the vector $base^w[1 \ldots l_{\max}]$ such that $base^w[j]$ is the smallest value $V^w(c)$ for all codes c such that $l_c = j$. Define the vector $disp[1 \ldots l_{\max}]$ such that $disp[j]$ is the number of codes c for which $l_c < j$. The index of code c of length l_c is:

$$\frac{V^w(c) - base^w[l_c]}{2^{w-l_c}} + disp[l_c]. \tag{1}$$

If the length of c is known, its index is given by that equation. Given the index, a computed branch would jump to the implementation of the virtual instruction.

To show examples of opcode frequencies, independently of a specific instruction set and samples, assume the n probabilities p_i of a special case of Zipf's law: $p_i = 1/(i H_n)$, $1 \leq i \leq n$, where H_n is the nth harmonic number $\sum_{j=1}^{n}(1/j)$. Such probabilities model well the static frequency of instructions in programs. Table 1 presents vectors $base^w$ and $disp$ for the Zipf-200 opcodes partly listed in Table 2. Their average length[2] is 6.0267.

3.1. *Very compact but slow decoding*

Assume that the beginning of an instruction is left justified in a variable `rd`. According to Eq. (1), decoding the opcode can be reduced to finding its length which can be done by a sequential search in *base*. Fig. 3 shows a fragment of C code for this slow but very compact decoder: Line 2 does the sequential search; the index of the code is calculated in `crd` by line 3 using (1); line 4 removes the opcode; line 5 does the actual branching to the virtual instruction implementation (using `gcc`'s computed `goto`).

[2] The average length is $\sum_{1 \leq i \leq n} l_{c_i} p_i$ where the opcode for the probability p_i is c_i and its length is l_{c_i}.

```
1  i = l_max;
2  while (rd < base_w[i]) i--;
3  crd = (rd-base_w[i] >> w-i) + disp[i];
4  rd <<= i;
5  goto *adr[crd];
```

Fig. 3. C code for a very compact, but slow, decoder for canonical ascending Huffman codes.

Table 1
The vectors `base_w` (aka $base^w$) and `disp` (*disp*) for Zipf-200

i	*disp*	$base^w$
3	1	$000{\cdot}2^{w-3}$
4	2	$0010{\cdot}2^{w-4}$
5	5	$01010{\cdot}2^{w-5}$
6	9	$011100{\cdot}2^{w-6}$
7	16	$1000110{\cdot}2^{w-7}$
8	33	$10101110{\cdot}2^{w-8}$
9	65	$110011100{\cdot}2^{w-9}$
10	135	$1110111110{\cdot}2^{w-10}$

This is a very compact decoder since its code is small and the vectors `base_w` and `disp` only contain $l_{\max}$ elements each. For Zipf-200 on a 32 bit processor, $l_{\max} = 10$ and $w = 32$, so the two vectors use a total of 80 bytes. Even for Zipf-400, that is 400 opcodes, a mere eight more bytes are needed.

But in general, this search is way too slow. The next section shows a better approach flexible in space and in speed.

4. Fast decoding

To increase speed, the linear search for the length of the opcode should be eliminated. This is done by a table look-up using the leftmost k bits of `rd`. The table contains branching addresses at which either decoding continue or the virtual instruction is emulated.

For the table look-up on k bits, three situations can arise:

(1) The opcode is recognized.
(2) The opcode is not recognized but its length is known.
(3) The opcode is not recognized and its length is unknown.

Case 1 is ideal, which occurs for all opcodes c where $l_c \leq k$. A direct jump is done to the implementation of the virtual instruction. In case 2, the length of the opcode is used to compute its index by Eq. (1); then a jump to the implementation of the virtual instruction is done. In case 3, the next bits are used to continue decoding using another look-up. Thus, the decoder has a tree structure where each interior node is case 3, simply called type 3

Table 2
Zipf-200 and timing for two decoders

i	opcode	Tree D_1		Tree D_2	
		ν	Time	ν	Time
1	000	a	7	a	7
2	0010	a	7	a	7
3	0011	a	7	a	7
4	0100	a	7	a	7
...	...				
15	100010	a	7	a	7
16	1000110	b	17	a	7
17	1000111	b	17	a	7
...	...				
31	1010101	b	17	a	7
32	1010110	e	14	a	7
33	10101110	e	14	a	7
34	10101111	e	14	a	7
35	10110000	b	17	a	7
...	...				
62	11001011	b	17	a	7
63	11001100	d	14	a	7
64	11001101	d	14	a	7
65	110011100	d	14	b	17
66	110011101	d	14	b	17
68	110011111	d	14	b	17
69	110100000	b	17	b	17
...	...				
124	111010111	b	17	b	17
125	111011000	c	14	b	17
126	111011001	c	14	b	17
...	...				
135	1110111110	c	14	b	17
136	1110111111	c	14	b	17
137	1111000000	b	17	b	17
138	1111000001	b	17	b	17
...	...				
199	1111111110	b	17	b	17
200	1111111111	b	17	b	17

nodes. In case 1 and 2 we have leaf nodes, simply called type 1 and 2 nodes. Note that each type 3 node requires a vector of addresses of its own, whereas type 2 nodes share the same vector.

In general, type 3 nodes do not use the same number of k bits to do a table look-up. For a node ν of type 3, k_ν is the number of bits used to do the table look-up. In particular, k_r denotes the number of bits used by the root r of a decoder.

Nodes of types 2 and 3 consume some CPU time. The time spent in a node of type i is denoted t_i. Note that $t_1 = 0$ because no further decoding is needed for type 1 nodes. These timing values do not have to correspond to any real unit of time, but simply be relative

to a known base value. For example, they could be approximated by the number of host processor cycles used at each node.

We denote by $T(D)$ the total time of a decoder D. It is the weighted sum of all decoding paths for all codes. More precisely, let C represents the set of codes decoded by D, P_c the set of nodes in the path from the root to the leaf of code c, f_c the frequence of code c, and $t_{\overline{v}}$ (i.e. t_1, t_2, or t_3) the execution time of node v. Then

$$T(D) = \sum_{c \in C} f_c \sum_{v \in P_c} t_{\overline{v}}. \tag{2}$$

To evaluate the space taken by the decoder, three constants are used: s_a is the number of bytes of an address (e.g. 4); s_2 is the number of bytes used by the machine code implementing a type 2 node and s_3 is for a type 3 node. We therefore take into account the space for look-up tables and the code to implement the decoding.

The total space taken by a decoder D is denoted by $S(D)$. More precisely, let k_v represent the number of bits used by the index of the look-up table of node v of type 3, and $\overline{v}$ the type of node v. Then the space taken by a node v is

$$s(v) = \begin{cases} 0 & \text{if } \overline{v} = 1 \\ s_2 & \text{if } \overline{v} = 2 \\ 2^{k_v} s_a + s_3 & \text{if } \overline{v} = 3. \end{cases} \tag{3}$$

The total space of a decoder D is

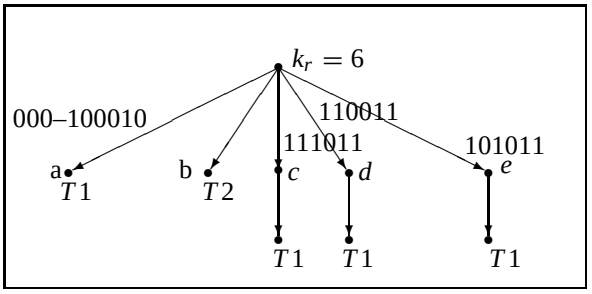
$$S(D) = \sum_{v \in D} s(v). \tag{4}$$

For example, Fig. 4 presents two decoder trees D_1 and D_2 for Zipf-200. They were automatically generated by our tool. Decoder D_1 does, at the root, a table look-up using six bits, and has three internal nodes doing table look-ups using four, three and two bits; whereas decoder D_2 does, at the root, a table look-up using eight bits and has one type 2 node. Note that there are opcodes of up to ten bits, but no table look-up is done using that many bits. The total space for decoder D_1 is 563 bytes and for D_2 it is 1084 bytes. The average decoding time for D_1 is 15.93 and for D_2 it is 13.8.

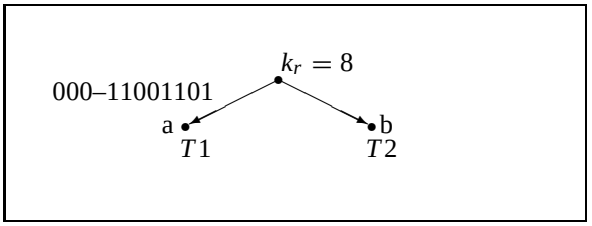
In Table 2 each opcode is shown along with the final node of decoding by the two decoders and corresponding relative time.

Given a space constraint, the basic parameters s_i and t_i, and the (static or dynamic) frequencies of the opcodes, we generate the fastest decoder. A branch and bound algorithm to do so is presented in [18]. It searches from the fastest to the slowest decoders, pruning its search using the fastest found decoder so far, and when the space constraints are met, it stops. For all our experiments, it takes a few seconds to find the fastest decoder.

To construct the decoder structure, the algorithm is general enough to accept static or dynamic (run-time) opcode frequencies. Dynamic frequencies are harder to obtain as they not only depend on the program samples but also on the input data of those programs. It is up to the designer of the virtual machine to assess the accuracy and relevance of dynamic frequencies and use them when they greatly differ from the static ones. On the other hand,



Tree D_1, $S(D_1) = 563$, $T(D_1) = 15.93$



Tree D_2, $S(D_2) = 1084$, $T(D_2) = 13.8$

Fig. 4. Two decoder trees D_1 and D_2 for Zipf-200, generated using the parameters $s_a = 4$, $s_2 = 30$, $s_3 = 25$, $t_2 = 10$, $t_3 = 7$. We have $k_c = 4$, $k_d = 3$ and $k_e = 2$.

Gregg and Waldron [11] have shown that dynamic frequencies are not that useful compared to the static ones.

5. The decoder C code

Fig. 5 shows the general structure of the C code for canonical decoders. All mathematical terms, such as $(w - k_r)$, become constants in the generated C code; for this term, w is the number of bits of `rd` and k_r is the number of bits for the index used at the root of the decoder for the table look-up. Similarly a term as complex looking as $base(C^{t2})_i + disp(C^{t2})_i$ becomes a constant since it can be computed statically.

Decoding begins at label `L_decode`. There is a label `L_`i for each case where more than one opcode of length i is not directly recognized by a node of type 3. These are type 2 nodes. There is a label `Lp_`*prefix* for each node of type 3, where *prefix* corresponds to the prefix of all codes for that node. For each virtual instruction *mne* the label `I`*mne* is the entry point of its implementation.

Line 1 loads, if necessary, some additional bytes in `rd`. The exact C code for this depends on the form of memory access used as discussed in Section 6. The incoming bits are justified in the high part of `rd` and `nb_rd` is adjusted to contain the number of bits in it. It always loads a multiple of eight bits, since the program counter points to a byte in memory, but `rd` does not necessarily contain a multiple of eight valid bits. Fig. 6 presents

```
L_decode:
1  {Transfer bytes from program to rd
    such that it has at least l_max bits,
    and increase nb_rd accordingly. }
2  crd = rd >> (w − k_r);
3  goto *adr_[crd];
L_i : /* opcodes of length i (type 2) */
4  crd = rd >> (w − i);
5  goto *adr_inst[crd - base(C^t2)_i + disp(C^t2)_i];
Lp_prefix: /* sub-decoder (type 3) */
6  crd = rd >> (w − l_prefix − k_prefix);
7  goto *adr_prefix[crd-v(prefix)2^k_prefix];
Imne:/* C code for mne (type 1) */
8  { If mne has parameters, transfer them to p_i }
   /* eliminate opcode and parameters */
9  rd <<= (l_opcode+l_parm);
10 nb_rd -= (l_opcode+l_parm);
11 { C code to emulate mne }
12 goto L_decode;
```

Fig. 5. General C code of opcode decoders.

```
#define BYTE(i) (unsigned int)prgm[pc+i]
rd    |= (BYTE(0) << 24 | BYTE(1) << 16
          | BYTE(2) << 8 | BYTE(3)) >> nb_rd;
pc    += (32 - nb_rd) >> 3;
nb_rd += (32 - nb_rd) & ~7;
```

Fig. 6. A simple technique for line 1 of Fig. 5.

a simple and inefficient portable implementation for line 1, for $w = 32$. Section 6 presents better portable techniques.

Line 2 is the root of a decoder where the first look-up is done; line 3 jumps to a type 2 or 3 node, or to the emulation of a virtual instruction. $w - k_r$ is a constant. At line 5, the term $base(C^{t2})_i + disp(C^{t2})_i$ is a constant: $base(C^{t2})_i$ is the ith value of $base^w/2^{w-i}$ but where $base^w$ is defined using only the codes C^{t2}, that is all codes treated by type 2 nodes. Using this subset of C might very well decrease the length of vector `adr_inst`. To be more precise, all addresses of virtual instructions in `adr_` are not duplicated in `adr_inst`. They also do not appear in any vectors `adr_`*prefix* for type 3 nodes. The vector $disp(C^{t2})$ is the corresponding vector of $base(C^{t2})$. Line 5 necessarily jumps to a virtual instruction. In

line 6, the term $w - l_{prefix} - k_{prefix}$ is a constant, l_{prefix} being the length of the prefix and k_{prefix} the number of bits decoded by this node. So the shifting `rd >>` $(w - l_{prefix} - k_{prefix})$ leaves in `crd` not only the k_{prefix} bits to decode but also the previous l_{prefix} bits. Line 7 applies the proper adjustment using the term $v(prefix)2^{k_{prefix}}$, which is the extra value left in `rd` before this node. This avoids shifting some bits out of `rd` until the end of decoding.

At line 8, decoding is complete and this is the emulation of the virtual instruction *mne*. If *mne* has some parameters, they are obtained here. This may use up all bits in `rd` or just part of them; it may also access memory. In most cases, bits should transit through `rd`. What lines 9 and 10 say, which is done differently depending on memory access forms (see Section 6), is that `rd` should contain the following bits and `nb_rd` should be maintained accordingly.

Finally, line 12 returns to the beginning of the decoding cycle. Again, this depends on the form of memory access as presented in Section 6. It could return to a point in the block of line 1 where it loads a specific number of bytes according to the number of bits consumed by *mne*.

6. Prefetching of code

One important part of the decoder C code was left unspecified, namely line 1, which loads bytes from memory into `rd`. Fig. 6 presents one possible simple implementation for line 1, where $w = 32$, but it was quickly discovered to be very inefficient. We investigated several other portable ways, three of which are reported in this section.

Getting opcodes and operands from memory into `rd` can be time consuming since multiple byte loads and bit manipulation operations are possibly needed. We have explored three different techniques to access memory. The first one, form-a, is simple, but shows major slowdowns on many benchmarks. The other two, form-b and form-c, show competitive speed; form-c being often faster than form-b but using more space for the interpreter. Our algorithm to generate decoders provides the option of using one of these three forms. Benchmarks in Section 7 show their relative merits.

The different prefetching methods are not used to mitigate the lack of caches or reduce the number of bytes read from memory. They are used to: reduce the number of instructions, in particular conditional branch instructions, for deciding how many bytes from memory need to be read to decode the next opcode; or reduce the number of merging operations, which requires executing several instructions, with `rd`.

For all forms, enough bits are in `rd` to go through the decoding tree, that is decode any opcode through the multiple level decoder, without accessing memory. This can simply be done by having at least $l_{\max}$, the length of the longest opcodes, valid bits in `rd`.

6.1. Simple form (form-a)

This version loads, from memory in `rd`, as many bytes as possible without shifting out, to the left, valid bits from it. It uses the number of valid bits in `rd` to load the minimum number of bytes necessary to maintain between $w - 7$ and w valid bits in `rd`. This can be done using a case analysis based on the value of `nb_rd`, reading from memory the required

bytes, shifting them to the left, and merging them to rd. The number of bytes to read is $\lfloor (w - \texttt{nb\_rd})/8 \rfloor$ and the number of bits to shift to the left is $(w - \texttt{nb\_rd}) \bmod 8$.

The advantage, compared with the code of Fig. 6, is a reduced number of memory accesses, bitwise 'or' operations, and shiftings.

The disadvantage is several comparisons to branch to the code for loading the appropriate number of bytes; and there are more mergings than form-c (see below) since more cases of one byte merging occurs.

6.2. Several-roots form (form-b)

In this form, as in the previous form-a, there are between $w - 7$ and w bits in rd at the beginning of the decoding tree. But instead of one entry point with complex verification of the number of bytes to load, there are several entry points r_x to the root of the decoder each one loading either x or $x + 1$ bytes. The implementation of these two cases is faster than the general selection of form-a since a single simple comparison is enough.

This form is possible, since each virtual instruction knows the number of bits extracted from rd (at lines 9 and 10), so that it almost knows the number of bytes to load in rd after its emulation. Indeed, suppose that a virtual instruction uses $b \leq w - 7$ bits, including its opcode. At the entry of its implementation there are between w and $w - 7$ bits in rd, therefore there are, after its emulation, between $w - b$ and $w - b - 7$ bits remaining in rd. So, there are between (A) $\lceil (b-1)/8 \rceil$ and (B) $1 + \lceil (b-1)/8 \rceil$ bytes to load in rd. If b is a constant, that is the instruction has a fixed length, which is a common case in practice, it is possible to jump to the proper root r_x without any test or computation. If b is not a constant, the virtual instruction implementation has to do some computation and test the number of bits left in rd anyway. This value is used to branch to the proper root r_x of the decoder. In the case where $b > w-7$, the virtual instruction itself has to load bytes from memory, thus also knows, after its emulation, the exact number of bytes to load. Note that no dynamic test is done to verify between cases (A) and (B), if b is a constant: it is hardcoded in the implementation of the interpreter. Otherwise, that is for a variable length instruction, some run-time tests should be done to branch to the proper root r_x of the decoder.

In some way, the proper number of bytes to load falls back to each virtual instruction which simply branches to one of the roots that does one integer relational test between a constant and nb_rd.

This is the advantage of that method compared to form-a: there is no need to compute the value $\lfloor (w - \texttt{nb\_rd})/8 \rfloor$ to know the number of bytes to load, and then to branch to the proper case which needs several comparisons; when the instruction is of fixed length, a simple single comparison is enough for form-b. But compared to form-c (see below) it still suffers from many small mergings of one byte.

The disadvantage of this method is a slightly bigger decoder due to multiple decoder roots.

6.3. Conditional form (form-c)

In this form, memory is accessed at the root, if and only if nb_rd is less than $l_{\max}$, the length of the longest opcodes. This value ensures that the tree decoder itself does not have to access memory to decode any opcode. If nb_rd is less than $l_{\max}$, as many bytes from

memory as possible are merged to `rd`. This means that access to memory is delayed as much as possible.

For example, if $l_{\max} = 14$, $w = 32$, and `nb_rd` $= 6$, three bytes are loaded and merged to `rd`; if `nb_rd` $= 15$ no bytes would be merged to `rd`.

The main advantage of this method is a reduced number of merging operations to `rd`. As a matter of fact, experiments show that this is in most cases the fastest approach.

The disadvantage is a larger interpreter, since if the virtual instruction uses more than $l_{\max}$ bits, for its operands, it is necessary to verify if there are enough bits in `rd` to access them. This case occurs less frequently in form-b for which there are $w - 7$ bits in `rd` after decoding the opcode (assuming $l_{\max} < w - 7$).

7. Experimental results

In order to evaluate our approach, we applied it to the Java Virtual Machine (JVM) on ten benchmarks [1] and the entire JDK 1.0.2 library; to the Scheme language on seven benchmarks and the R^4RS library; and to six synthetic benchmarks to demonstrate the worst case scenarios.

For all benchmarks two processors are used: a 600 MHz Pentium III and a 200 MHz Sparc Ultra-1 with respectively 32 KB and 1 MB level 1 cache. All C programs were compiled using `gcc` version 2.8.1 for SunOS and version 2.91.66 for Linux with the same optimizing parameter, namely `-O3`.

7.1. Java benchmarks

We use the Java Virtual Machine to demonstrate our approach on a widely available bytecode using Harissa [22]. Most virtual instructions' implementations are unchanged but branching instructions are modified to branch on non-byte boundaries. Harissa uses a C switch statement to decode bytecode instructions. All cases of this switch are transformed into C macro-instructions and are used by the tool to automatically generate the interpreter of the compressed code that is the implementation of macro-instructions and instructions that use new formats. The switch is removed and replaced by an opcodes decoder automatically generated from our tool.

The training set is the file `classes.zip`, i.e. the set of libraries from JDK 1.0.2, containing over 400 class files. The total bytecode size is 270932 bytes. Note that the benchmarks (see below) were not used as a training set. This may represent an embedded design where the standard libraries and the virtual machine are placed in ROM but where executed programs are downloaded in RAM. In this experiment, the benchmarks would take the roles of downloaded programs.

After training, the resulting shortest opcode has three bits and the longest opcodes have twelve bits. Forty of the existing instructions were duplicated but with shorter parameter fields, resulting in a JVM machine of 241 instructions. This extension was done automatically by our tool to generate virtual instruction sets from samples of programs [17]. The sole choices of macro-instructions and parameter lengths were done to better compress the library classes and not for speed.

Table 3
Absolute execution time, in seconds, for the original (i.e. non-compressed) Java benchmarks with modified Harissa JVM on Pentium and SPARC; and the size of the original bytecodes

Benchmark	Absolute time		Code size
	Pentium	SPARC	in bytes
NeuralNet	27.8	46.64	7467
FPemulation	3.82	5.29	3724
IDEAencryption	5.40	6.46	1800
Assignment	1.49	2.42	1634
LUdecomposition	3.29	4.60	1602
StringSort	7.68	10.35	1541
Huffman	2.50	3.98	1395
BitfieldOps	5.11	6.21	833
NumericSort	2.75	3.99	773
Fourier	1.83	2.24	640

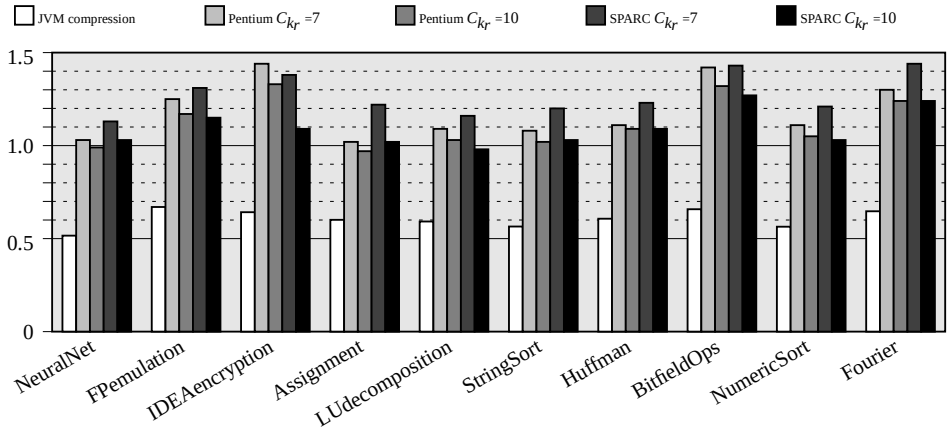


Fig. 7. Relative speed and compression factors (i.e. compressed code size/original code size) of Java benchmarks with modified Harissa JVM.

It took around twenty minutes of CPU time (on a 600 MHz Pentium III) to create the new instruction set (i.e. macro-instructions, new formats, and opcodes). It took less than ten seconds to generate the C code of the new JVM.

For the libraries, that is `classes.zip`, a 0.609 compression factor (i.e. compressed code size/original code size) is obtained.

Fig. 7 presents the timing results and the compression factors of bytecodes for the BYTEmark Java benchmarks [1]. The execution times and sizes of the non-compressed bytecodes are presented in Table 3. These are moderate size benchmarks suited to evaluate

the speed of JVM implementations. The compression factors take into account the compression of opcodes, the compact operands, and the use of macro-instructions. They do not take into account the decoder sizes or the increase or decrease in the interpreter size. As we discussed in the introduction, this is to separate the space taken by the virtual machine and the space taken by the virtual programs. The overall compression should take into account all programs loaded into the systems, the libraries needed to run them, and the size of the interpreter, including the new decoders. Obviously, this depends on the application used. If the application's overall size is 100 KB for the original bytecode, including the libraries, we may expect a saving of around 40%, that is 40 KB. The increased size of the decoder may be largely offset by this saving. Below, we give the size of the decoders used to decode compressed programs compared with the uncompressed decoder.

The speed is relative to the execution of the uncompressed original bytecode on the original Harissa JVM. Note that all these programs execute some part of the JDK libraries `classes.zip`. Slight speedups are observed: they are mainly due to the inlined macro-instructions which increase speed of execution.

We use memory access form-c with two decoders having the following structures: (1) $k_r = 7$, five nodes of type 2, namely L_{8-12}, and three nodes of type 3, all directly below the root; the sum of table sizes is $(2^7 + 5 \times 2^4 + 3 \times 2^5) \times 4 = 1216$ bytes; (2) $k_r = 10$, two nodes of type 2, namely L_{10-11}, and one node of type 3; the sum of table sizes is $(2^{10} + 2 \times 2^4 + 2^2) \times 4 = 4240$ bytes. Assuming that the original bytecode decoder could use a simple flat look-up table of 256 entries of four bytes each, its size would be 1024 bytes. Therefore, for $k_r = 7$ the size of the tables for the decoder increases by 192 bytes; for $k_r = 10$ it increases by 3216 bytes. If the space of the virtual machine is a strong concern, the $k_r = 7$ decoder adds a very small amount of space to the overall virtual machine; whereas, if the virtual machine is stored, let's say in ROM, for which sufficient space would still be available, it would be better to use the $k_r = 10$ decoder to increase speed.

The SPARC processor shows the best average slowdown of 9.3% for $k_r = 10$. One advantage of the SPARC is a larger number of registers available compared to the Pentium. The overall speed is sensitive to register availability, since the interpreter frequently accesses the variables `rd`, `pc`, and `nb_rd`. These should be kept in registers to have good performance. The Pentium assembly code reveals that not enough registers are available to do that.

The worst speed results are the Fourier and Bitfieldops benchmarks. This is due to the frequent execution of instructions having long opcodes and small granularities (i.e. the amount of processing done by the instructions). Some of them are floating-point virtual instructions, not statically frequent in `classes.zip`. They also do not access object fields as frequently as the other benchmarks. Since the `getfield` and `putfield` instructions have a moderate granularity, they increase execution time compared to decoding. On the other hand, Assignment, StringSort, NeuralNet, NumericSort, and LUdecomposition show a small slowdown.

The benchmarks Assignment, StringSort, and NeuralNet have a large number of virtual method calls as well as field accesses. As mentioned, field accesses hide decoding overhead, and this is also true for method invocation, be it static or virtual. They show little slowdown for the SPARC with a good performance for the Pentium.

Table 4
Original (not compressed) code size of Scheme programs and their absolute execution time in seconds, on Pentium and SPARC

Program	Code size in bytes	Pentium	SPARC
libR4	32040	N/A	N/A
conform	28599	11.70	32.84
earley	26271	1.70	4.14
qsort	5827	1.83	5.54
destruct	3371	0.79	2.16
mm	2550	3.70	9.26
tak	582	1.55	4.35
fib	169	1.76	4.74

Half of the benchmarks have a 40% reduction in size with a negligible slowdown ($\leq$3%).

The work of Clausen et al. [5] presents the compression of JVM bytecodes compared with gzip. It is applied separately on each method. The gzip compression factors vary from 0.66 to 0.91. Our technique gives better compression factors.

7.2. *The* Scheme *language*

Our approach has also been applied to the Scheme language [17]. From a general stack machine called Machina our tools create a new set of instructions called Schemina. Machina has only 41 instructions.

This experiment is quite different compared to the JVM as it starts from a very simple and general virtual machine not tailored for Scheme. Due to the non-optimized bytecode encoding, the compression factors obtained are better than for the JVM. Essentially, more useful macro-instructions were discovered and they were longer.

The training set (i.e. the samples of programs) was a subset of the R4RS Scheme library (called libR4 in Fig. 9) and seven benchmarks. A total of 112 instructions were generated by our tools, including the 41 original ones.

Fig. 9 compares the compression factors of our technique with gzip applied to the original bytecode. The original code sizes are presented in Table 4. We only used it to compare compression performances since gzip encoding cannot be executed without prior decompression. gzip can have better performances for two major reasons: it compresses disregarding basic block boundaries, and it disallows non-sequential decompression.

In several cases our approach is close or better than gzip and we can still efficiently execute our compressed code.

Fig. 8 presents, relatively to the uncompressed original Machina programs, execution time of the compressed Scheme programs. The absolute times for the original (i.e. non-compressed) programs are presented in Table 4. For several benchmarks there are speedups since macro-instructions increase speed and many of the new instructions have short opcodes and operands.

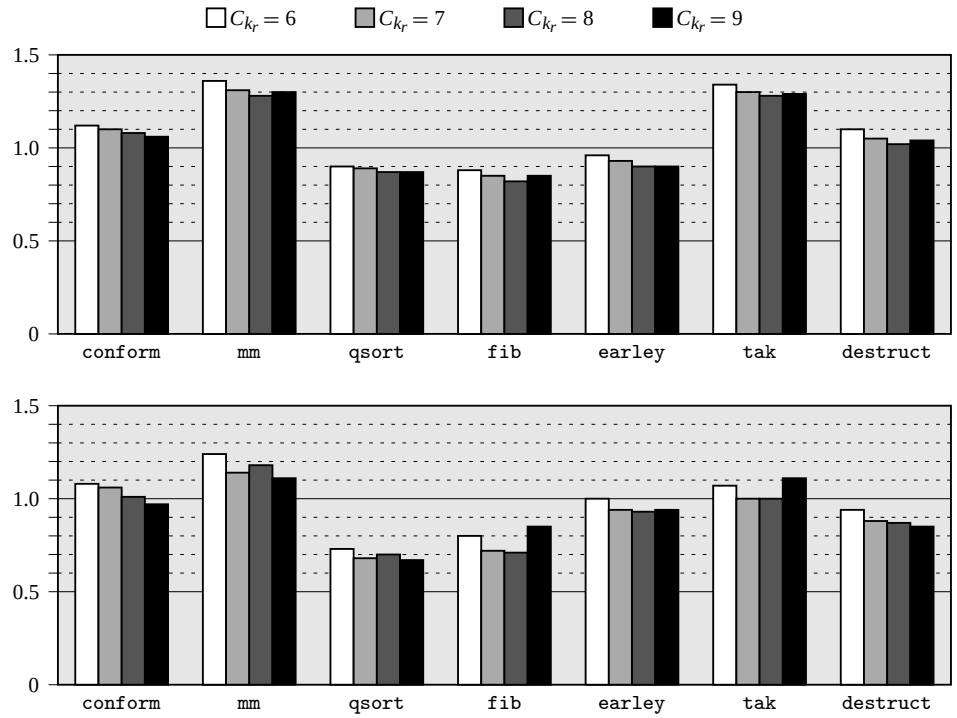


Fig. 8. Relative execution time of compressed Scheme programs, using form-c on Pentium (top) and SPARC (bottom) for several decoders.

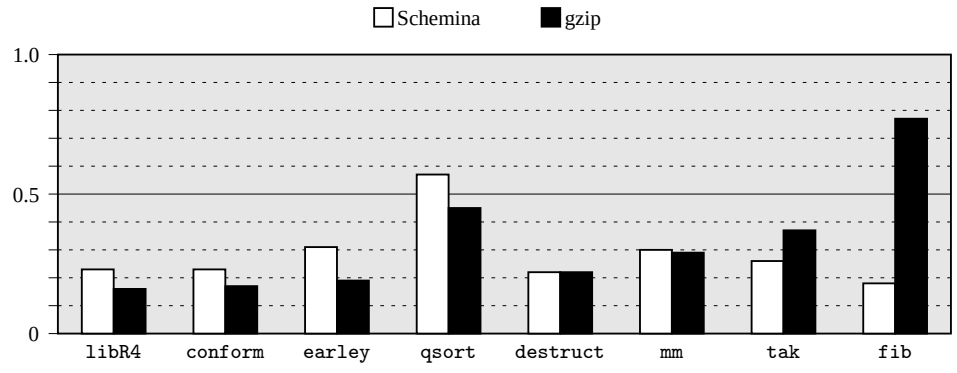


Fig. 9. Compression factors (i.e. compressed code size/original code size) for Scheme programs compared to gzip.

Table 5
Absolute time, in seconds, to execute uncompressed programs, based on Zipf-20

Pentium			SPARC		
M_1	M_2	M_3	M_1	M_2	M_3
0.38	0.45	0.81	2.13	2.56	5.08
M_4	M_5	M_6	M_4	M_5	M_6
0.40	0.49	0.85	2.32	2.83	3.76

7.3. Synthetic benchmarks

The Java and Scheme benchmarks demonstrate the applicability of the approach in a realistic setting. But it raises the question of hidden overhead by the emulation of the virtual instructions. Also, inlined macro-instructions increase speed. Therefore, we also present synthetic benchmark timings, where the frequency of instructions, their granularity, and their operand lengths are precisely defined; there are no macro-instructions used for these. In other words, the synthetic benchmarks more clearly show the overhead of Huffman decoding and non-byte alignment.

For the synthetic benchmarks, we use six virtual machines of different granularities allowing better measurement of decoding overhead. They all have twenty instructions, without parameter for the first three machines, but for the last three machines, six instructions have a parameter of length 2, 2, 3, 4, 5 and 7 bits. The opcodes are encoded based on Zipf-20 probabilities.

In the first machine, all twenty virtual instructions add one to an integer counter c_i; in the second machine each instruction does two additional integer operations; in the third one, each instruction does two additional memory accesses to simulate a stack. Machines 4–6 have parameters and do the same work as machines 1–3 respectively, but six instructions have parameters and add them to their own counter c_i. We use the same program for the six machines: it is a sequence of the twenty instructions, from instruction 1 to 20, performing 4×10^5 iterations; that is the last instruction does a jump to the first instruction which stops the execution when counter c_1 reaches this value. The opcodes are compressed based on the Zipf-20 probabilities which have an average length of 3.6 bits. Three decoders are applied on all six machines executed on two host processors.

An interpreter was used to decode the uncompressed programs. These programs are bytecoded: one byte for each opcode and two bytes for an operand, if applicable. The decoding is a computed branch, indexed by the opcode, to the virtual instruction implementation. Each implementation loads its operands, emulates the operation and jumps back to the beginning of the decoding cycle.

Fig. 10 presents the timing results for compressed programs, relative to the uncompressed ones. The absolute times are presented in Table 5. The simple memory access form-a is disappointing but form-b and form-c are good. The two forms are close in performance even though form-c is often the better. Since form-b generates more compact interpreters there is an informed trade-off to make.

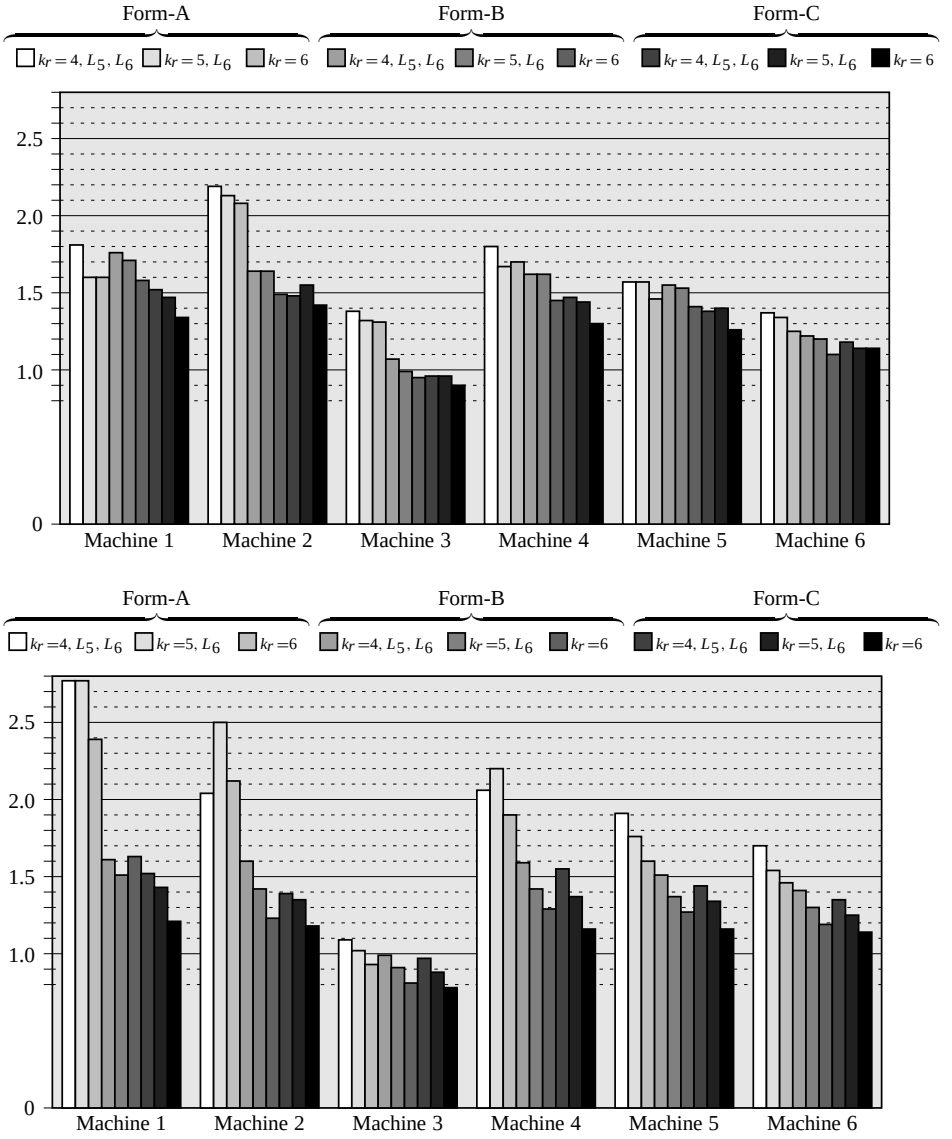


Fig. 10. Relative time to execute compressed programs, based on opcodes from Zipf-20 probabilities, for six virtual machines, three decoders, three memory access forms (form-a, form-b and form-c), on Pentium (top) and SPARC (bottom).

As expected, the best results are obtained for machine 3, since the instruction granularities hide some overhead of decoding. In particular, on Pentium and SPARC there is an acceleration for the parameterless instructions. This is due to the reduction in memory accesses and extraction of operands. With $k_r = 6$, decoding is done in one step, and most often the next opcode is in `rd`, which is, in these benchmarks, a processor register.

From this experiment we can conclude that even with virtual instructions doing almost no work, as in machine 1, and with a small decoder, the decoding is around a 50% overhead (as in form-c used with a $k_r = 4$ decoder). This is an extreme artificial setting aimed to demonstrate the worst case performance. On the other hand, if we have instructions with no parameter and enough granularity, a speedup can be observed. This is probably due to a reduced number of accesses to memory as `rd` is most of the time loaded with more than one instruction; this is not the case with the original bytecode. But we have not explicitly verified this hypothesis.

7.4. Summary of the experiments

In conclusion, for JVM, the average compression factor is around 0.6 for 400 classes of the JDK 1.0.2 and the ten benchmarks. For half of the benchmarks, the slowdown is hardly noticeable. This shows the practicality of the approach. The synthetic benchmarks show more explicitly the overhead of decoding our compressed bytecode, demonstrating that even a speedup can be achieved in some cases without macro-instructions. The Scheme results show that starting from a very general machine, our compression creates efficient and compact instructions. They also show that we can come close to gzip compression performance and still efficiently decode the compressed instructions.

8. Related work

The compression of bytecodes for virtual machines was addressed by Wilner for the SDL language on Burroughs B1700 [29]. It uses Huffman codes but decoding was done bit by bit by the microcode. This is too slow to be practical at the software level.

Patterson and Henessy manually designed a compact native instruction set by studying sample programs generated from C code [23].

Decoding of Huffman encoded instructions has also been studied at the hardware level by several researchers [16,19,2]. They usually decompress between the memory and the instruction cache. They do not use fast decoding methods applicable at the software level.

Ernst et al. [9] compress native code by generating a tailored VM, using macro-instructions and fixing parameters, from the intermediate form emitted by a C compiler. It is similar to Proebsting's work [24]. Their technique is competitive with gzip on native code. But it is not reported if the compression obtained is due to the use of the VM or the compression of the virtual program. Moreover, no timing of the execution of compressed programs is reported.

Cooper and McIntosh [6] reduce program size by replacing particular repetitive sequences of instructions with a branch. The code saving is on average 5%. Cooper et al. [7] searches, using a genetic algorithm technique, a combination of compilation techniques to reduce code size. These works differ from ours since they are done on native code and no Huffman encoding and argument compacting are applied.

Pugh [25] applies several techniques to compress Java class files. This work differs from ours since decompression is performed before execution.

The work of Rayside et al. [26] also applies to class files, but these techniques do not apply to the bytecode itself.

Hoogerbrugge et al. [12] uses a similar strategy of the Thumb and MIPS16 processors [28,15] to compress some parts of the program. But instead of applying compression on the binary executable, they automatically generate a tailored virtual machine for the intermediate form of the C program. When the intermediate form is translated into a virtual program, frequent sequences of virtual instructions are replaced by one opcode. This particular technique gives a 30% reduction in size compared to the virtual program. Our work is complementary by further reducing the size of the virtual programs using compressed virtual instructions.

Lucco [20] applies compression to x86 native code using a dictionary technique to keep track of repeated short sequences of instructions. At least one decompression is performed before the execution of a basic block, requiring a buffer space to keep the decompressed copy. Our work differs as we apply it to the context of virtual machines and directly decode compressed instructions.

Clausen et al. [5] compresses bytecode by replacing repetitive sequences of JVM instructions by macro-instructions. They obtain an average compression factor of 0.85 with a slowdown from 19% to 27%.

The work of Evans and Fraser [10] has an identical goal as ours: direct execution of compressed bytecode. The compression technique is based on the modifications of a grammar of the bytecode. Their technique avoids variable length instructions contrary to our technique. Good compression factors on large programs are obtained although no execution times are reported. This technique could be combined with ours to cover a larger range of program sizes; but, we believe, to the detriment of execution speed.

Debray and Evans [8] use canonical Huffman code on binary executables, but using the slow decoding technique as in Section 3.1. They avoid compression on frequently executed parts of the code to obtain reasonable execution speed.

In general, our approach differs from these previous approaches as we interpret compressed virtual instructions directly without an explicit partial or whole decompression, avoiding using any additional RAM, and moreover it is done over variable bit-length encoding, specifically Huffman encoding of operational codes.

9. Summary

This work has shown that decoding canonical Huffman encoded opcodes, at the software level, in the context of virtual instructions, can be done efficiently. The speed of decoding increases with the size of the decoder. A general structure of compact decoders has been shown to be effective, permitting a gradual trade-off between speed of decoding and space constraints.

Huffman decoding is not the only difficulty for quickly interpreting compressed virtual instructions, memory access for variable length bit fields is also important. Two prefetching techniques were shown to achieve good results.

The efficiency of the decoders has been demonstrated on simple synthetic benchmarks, on the Scheme language, and on Java benchmarks showing an average slowdown ranging from 2% to 27% depending on the processor and the size of the decoders. Half of the Java benchmarks have a 40% reduction in size with a negligible slowdown ($\leq$3%).

References

[1] Java BYTEmark benchmarks: source code and results. http://www.igd.fhg.de/~zach/benchmarks, 1999.
[2] M. Benes, S.M. Nowick, A. Wolfe, A fast asynchronous Huffman decoder for compressed-code embedded processors, in: Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, September 1998.
[3] J.P. Bennet, G.C. Smith, The need for reduced byte stream instruction sets, The Computer Journal 32 (1989) 370–373.
[4] Y. Choueka, S.T. Klein, Y. Perl, Efficient variants of huffman codes in high level languages, in: Proc. of the 8th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval, June 1985, pp. 122–130.
[5] L.R. Clausen, U.P. Schultz, C. Consel, G. Muller, Java bytecode compression for low-end embedded systems, ACM Transactions on Programming Languages and Systems 22 (3) (2000) 471–489.
[6] K.D. Cooper, N. McIntosh, Enhanced code compression for embedded RISC processors, in: SIGPLAN Conference on Programming Language Design and Implementation, 1999, pp. 139–149.
[7] K.D. Cooper, P.J. Schielke, D. Subramanian, Optimizing for reduced code space using genetic algorithms, ACM SIGPLAN Notices 34 (7) (1999) 1–9.
[8] S. Debray, W.S. Evans, Profile-guided code compression, in: Proc. Conf. on Programming Languages Design and Implementation, 2002, pp. 95–105.
[9] J. Ernst, C.W. Fraser, W. Evans, S. Lucco, T.A. Proebsting, Code compression, in: Proc. Conf. on Programming Languages Design and Implementation, June 1997, pp. 358–365.
[10] W.S. Evans, C.W. Fraser, Bytecode compression via profiled grammar rewriting, in: Proc. Conf. on Programming Languages Design and Implementation, 2001, pp. 148–155.
[11] D. Gregg, J. Waldron, Primitive sequences in general purpose Forth programs, in: M.A. Ertl (Ed.), 18th EuroForth Conference, 2002, pp. 24–32 (Refereed).
[12] J. Hoogerbrugge, L. Augusteijn, J. Trum, R. van de Wiel, A code compression system based on pipelined interpreters, Software—Practice and Experience 29 (11) (1999) 1005–1023.
[13] D.A. Huffman, A method for the construction of minimum redundancy codes, in: Proc. IRE, vol. 40, September 1952, pp. 1098–1101.
[14] T.M. Kemp, R.M. Montoye, J.D. Harper, J.D. Palmer, D.J. Auerbach, A decompression core for PowerPC, IBM Journal of Research and Development 42 (6) (1998).
[15] K. Kissell, MIPS16: High-density MIPS for the Embedded Market, Silicon Graphics MIPS Group, 1997.
[16] M. Kozuch, A. Wolfe, Compression of embedded system programs, in: Proc. Int'l Conf. on Computer Design, 1994, pp. 270–277.
[17] M. Latendresse, Automatic generation of compact programs and virtual machines for Scheme, in: M. Felleisen (Ed.), Proceedings of the Workshop on Scheme and Functional Programming, September 2000. Available at http://www.iro.umontreal.ca/~latendre/publications/.
[18] M. Latendresse, M. Feeley, Fast and compact decoding of Huffman encoded virtual instructions, Technical Report DIRO-1219, University of Montreal, November 2002. Available at http://www.iro.umontreal.ca/~latendre/publications/.
[19] H. Lekatsas, W. Wayne, Code compression for embedded systems, in: Design Automation Conference, 1998, pp. 516–521.
[20] S. Lucco, Split-stream dictionary program compression, in: Proc. Conf. on Programming Languages Design and Implementation, Vancouver, British Columbia, 2000, pp. 27–34.
[21] A. Moffat, A. Turpin, On the implementation of minimum redundancy prefix codes, IEEE Transactions on Communications 45 (10) (1997) 1200–1207.
[22] G. Muller, B. Moura, F. Bellard, C. Consel, Harissa: A flexible and efficient Java environment mixing bytecode and compiled code, in: Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems, 16–20 June, Usenix Association, Berkeley, 1997, pp. 1–20.
[23] D. Patterson, J. Hennessy, Computer Architecture, a Quantitative Approach, 2nd edition, Morgan Kaufmann, 1996.
[24] T.A. Proebsting, Optimizing a ANSI C interpreter with superoperators. in: Proc. Symp. on Principles of Programming Languages, 1995, pp. 322–332.

[25] W. Pugh, Compressing Java class files, in: Proc. Conf. on Programming Languages Design and Implementation, 1999, pp. 247–258.
[26] D. Rayside, E. Mamas, E. Hons, Compact java binaries for embedded systems, in: Cascon, November 1999, pp. 1–14.
[27] E.S. Schwartz, B. Kallick, Generating a canonical prefix encoding, Communications of the ACM 7 (3) (1964) 166–169.
[28] J.L. Turley, Thumb squeezes ARM code size, Microprocessor Report 9 (4) (1995) 1–6.
[29] W.T. Wilner, Burroughs B1700 memory utilization, AFIPS FJCC 41 (1972) 579–586.