Generation of Fast Interpreters for Huffman Compressed Bytecode

Mario Latendresse
Northrop Grumman IT/
Technology Advancement Group
FNMOC/U.S. Navy
7 Grace Hopper, Monterey, CA, USA
mario.latendresse.ca@metnet.navy.mil

Marc Feeley
Département d'informatique et
recherche opérationnelle
Université de Montréal
C.P. 6128, succ. centre-ville
Montréal, H3C 3J7, Canada
feeley@IRO.UMontreal.CA

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%. Compression factors highly depend on the original bytecode and the training sample, but typically vary from 30% to 60%.

Categories and Subject Descriptors

D.3 [**Software**]: Programming Languages; D.3.4 [**Programming Languages**]: Processors—*Interpreters*

Keywords

Code compression, canonical Huffman code, decoder, Java

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 [5, 13, 12].

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.

Some researchers [9, 10, 3] have stated the possibility of using Huffman codes to compress bytecode, usually to conclude that this would, at the software level, increase decoding time to an unacceptable level or need too much space for table look-up.

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.

Our work shows 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.

The primary focus of this paper is to show that there are techniques to efficiently decode such compressed instructions.

For speed, canonical Huffman codes should not be decoded bit by bit; instead, blocks of k bits should be used. Such an idea has been explored previously by Turpin and Moffat [19]. We have extended their work to allow multiple k bits look-ups and generate decoders given a space constraint.

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

Figure 1 presents our general framework. The sample of programs is bytecode encoded with an unmodified compiler. An instruction set encoding to compactly represent the sample is then generated by a tool. This requires an analysis

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
IVME'03, June 12, 2003, San Diego, California, USA.
Copyright 2003 ACM 1-58113-655-2/03/0006 ...$5.00.

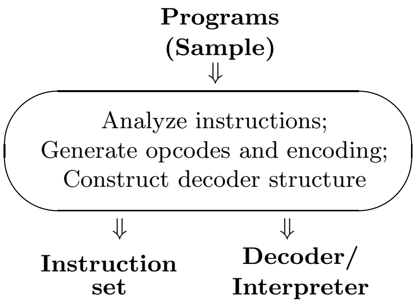


Figure 1: Creation of instruction set, its decoder and interpreter.

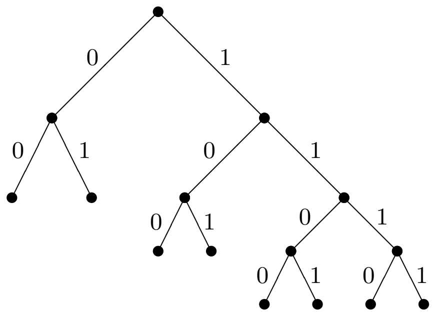


Figure 2: A canonical ascending Huffman tree.

of the instruction frequencies, the length of operands, etc. of the sample. 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 an encoding for them. These instructions are either macro-instructions to replace a sequence of instructions, or a basic instruction with a new format for the operands. We proceed as follows to create them. From the sample of programs:

1. A dictionary of (possibly overlapping) repetitive sequences is built. We limit their number by their length and a minimum of their occurrences in the sample.
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 to be able to encode all possible programs.
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 [15].

Henceforth, the following setting is used: the opcodes are variable length canonical Huffman codes generated using the static frequencies of the opcodes from a sample of 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 deplacements of branching instructions are in bits, but instructions following subroutine calls are byte-aligned—return addresses are in bytes.

3. HUFFMAN ENCODING OF OPCODES

We encode opcodes using canonical Huffman codes [24]. These are similar to Huffman codes built by the original bottom up method of [11], 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.

Figure 2 shows such a canonical tree, where branches are pushed to the right: it is an ascending tree, since it is possible to order the codes in increasing length and numerical value.

Let l_c be the length in bits of code c, $v(c)$ 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 bits 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 canonical Huffman codes, $l_{\max}$ their maximum length and w a constant 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:

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

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: $p_i = 1/(iH_n)$, $1 \leq i \leq n$, 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[1] 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 equation 1, decoding the opcode can be reduced to finding its length which can be done by a sequential search in *base*. Figure 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`).

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.

[1]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}.

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

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

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	$1110111110{\cdot}2^{w-10}$

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

4. FAST DECODING

To increase speed, the linear search for the length of the opcode must 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.

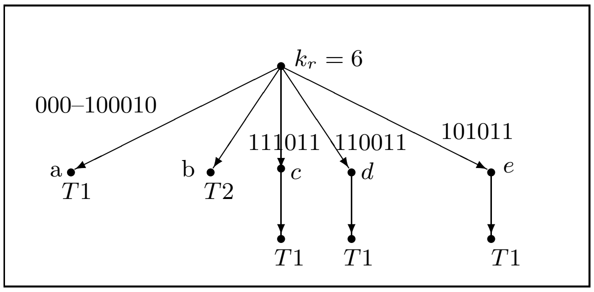
For the table look-up on k bits, three situations can arise:

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.

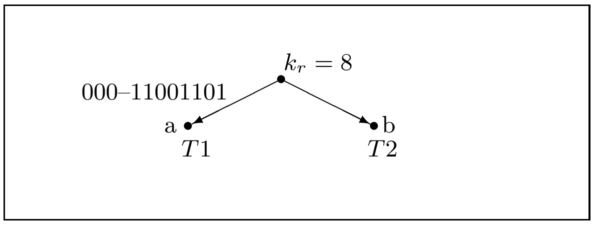
Case 1 is ideal, which occurs for all opcodes c where $l_c \leq k$. A direct jump is done to the implementation of the virtual instruction. In case 2, the length of the opcode is used to compute its index by equation 1; then a jump to the implementation of the virtual instruction is done. In case 3, 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 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, interior 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.

Each node requires some time to execute. 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.



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



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

Figure 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 $k_e = 2$.

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.

Figure 4 presents two decoder trees D_1 and D_2 for Zipf-200. Decoder D_1 does, at the root, a table look-up using 6 bits, and has three internal nodes doing table look-ups using 4, 3 and 2 bits; whereas decoder D_2 does, at the root, a table look-up using 8 bits and has one type 2 node. Note that there are opcodes of up to 10 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 [16]. It searches from the fastest to the slowest decoders 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 accurancy and relevance of dynamic frequencies and use them when they greatly differ from the static ones.

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

Table 2: Zipf-200 and timing for two decoders.

5. THE DECODER C CODE

Figure 5 shows the general structure of the C code for canonical decoders. 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. Figure 6 presents 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

```
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;
```

Figure 5: General C code of canonical 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;
```

Figure 6: A simple technique for line 1 of Figure 5.

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`. We investigated several 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. Our algorithm to generate decoders provides the option of using one of these three forms. Benchmarks in section 7 show their relative merits. For all forms, enough bits are in `rd`, at the root of the decoder, to decode one opcode without accessing memory.

6.1 Simple form (Form-a)

This version uses the number of bits in `rd` to load the minimum number of bytes necessary to maintain between $w-7$ and w bits in `rd` at line 2. This can simply 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 is $(w - \texttt{nb\_rd}) \bmod 8$.

This technique loads in `rd` as many bytes as possible. The advantages is a reduced number of merging operations and faster access to operands since they are most often hauled in before decoding the opcode. The other advantage is a smaller interpreter, since it reduces the number of instructions accessing operands in memory. The disadvantage is a slow operation at every cycle to verify and load the correct number of bytes.

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. But instead of one entry point with complex verification of the number of bytes to load, there are several entry points to the root of the decoder each one loading either x or $x+1$ bytes. The decision between case x and $x+1$ is faster than the general case of form-a.

It is faster, since each virtual instruction knows the number of bits extracted from `rd` (at lines 9 and 10), it knows approximately the number of bytes to load 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 $\lceil(b-1)/8\rceil$ and $1+\lceil(b-1)/8\rceil$ bytes to load in `rd`. If b is a constant, which is quite a common case in practice, it is possible to jump to the proper root r_x without any test. If b is not a constant, that is a variable length instruction, the virtual instruction implementation has to calculate the number of bits left in rd anyway. 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 the two cases, if b is a constant. It is hardcoded in the implementation of the interpreter.

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`. The disadvantage of this method is a slightly bigger decoder.

6.3 Conditional form (Form-c)

In this form, there is a verification of the number of bits in `rd` at the root of the decoder. Memory is accessed, at the root, if and only if `nb_rd` is under $l_{\max}$, the longest opcode. This ensures that the decoder does not access memory while decoding an opcode. If it is under $l_{\max}$, as many bytes from memory as possible are merged to `rd`. For example, if $l_{\max} = 14$, $w = 32$, and `nb_rd`$= 6$, three bytes are loaded and merged to `rd`. This means that access to memory is delayed as much as possible.

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

The disadvantage is that more virtual instruction implementations have to access memory for their operands.

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 $\mathrm{R}^4\mathrm{RS}$ library; and to six synthetic benchmarks to demonstrate the worst case scenarios.

For all benchmarks two processors are used: a 600MHz Pentium III and a 200MHz Sparc Ultra-1 with respectively 32KB and 1MB 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 [20]. Most virtual instructions' implementation are unchanged but branching instructions must be modified to branch on non-byte boundaries. Harissa uses a C switch statement to decode bytecoded instructions. All cases of this switch are transformed into C macro-instructions and are used by the canonical decoder to implement each instruction in the JVM machine for compressed code. The switch is removed and replaced by a decoder automatically generated from our tool.

Table 3 presents the timing results and the compression factors of bytecodes for the BYTEmark Java benchmarks [1]. These are moderate size benchmarks suited to evaluate the speed of JVM implementations. The compression factor is the length of compressed code divided by the uncompressed code (bytecode). It takes into account the compression of opcodes, the compact operands, and the use of macro-instructions.

The training set is the `classes.zip` from JDK 1.0.2, containing over 400 class files. 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 241 instructions JVM machine. This extension was done automatically by our tool to generate virtual instruction sets from a sample of programs [15]. The sole choices of macro-instructions and parameter lengths were done to better compress the classes and not for speed. All class files for the BYTEmark benchmarks, including all libraries in `classes.zip`, are compressed based

Benchmark	**Absolute Time Uncompressed** Pentium	SPARC	**Relative Time Compressed** Pentium $C_{k_r=7}$	Pentium $C_{k_r=10}$	SPARC $C_{k_r=7}$	SPARC $C_{k_r=10}$	**Compression Factor of JVM Code**	**Size of JVM code in bytes**
NumericSort	2.75	3.99	1.11	1.05	1.21	1.03	56.4%	773
StringSort	7.68	10.35	1.08	1.02	1.20	1.03	56.5%	1541
BitfieldOps	5.11	6.21	1.42	1.32	1.43	1.27	65.8%	833
FPemulation	3.82	5.29	1.25	1.17	1.31	1.15	67.0%	3724
Fourier	1.83	2.24	1.30	1.24	1.44	1.24	64.7%	640
Assignment	1.49	2.42	1.02	0.97	1.22	1.02	60.1%	1634
IDEAencryption	5.40	6.46	1.44	1.33	1.38	1.09	64.2%	1800
Huffman	2.50	3.98	1.11	1.09	1.23	1.09	60.7%	1395
NeuralNet	27.8	46.64	1.03	0.99	1.13	1.03	51.6%	7467
LUdecomposition	3.29	4.60	1.09	1.03	1.16	0.98	59.2%	1602
Average			**1.18**	**1.12**	**1.27**	**1.09**	**58.8%**	

Table 3: Relative speed and compression factors of Java benchmarks with modified Harissa JVM.

	Relative time Pentium					**Relative time** SPARC				
	$C_{k_r=6}$	$C_{k_r=7}$	$C_{k_r=8}$	$C_{k_r=9}$	$C_{k_r=10}$	$C_{k_r=6}$	$C_{k_r=7}$	$C_{k_r=8}$	$C_{k_r=9}$	$C_{k_r=10}$
fib	0.88	0.85	0.82	0.85	0.85	0.80	0.72	0.71	0.85	0.66
tak	1.34	1.30	1.28	1.29	1.29	1.07	1.00	1.00	1.11	0.88
earley	0.96	0.93	0.90	0.90	0.90	1.00	0.94	0.93	0.94	0.82
conform	1.12	1.10	1.08	1.06	1.01	1.08	1.06	1.01	0.97	0.95
mm	1.36	1.31	1.28	1.30	1.29	1.24	1.14	1.18	1.11	1.04
destruct	1.10	1.05	1.02	1.04	1.04	0.94	0.88	0.87	0.85	0.79
qsort	0.90	0.89	0.87	0.87	0.87	0.73	0.68	0.70	0.67	0.60

Table 4: Relative execution time of compressed Scheme programs, using form-c on Pentium and SPARC.

on the new Huffman opcodes, the new formats, and the macro-instructions. For `classes.zip`, a 60.9% compression factor is obtained and an overall average of 58.8% for the benchmarks.

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; 2) $k_r = 10$, two nodes of type 2, namely L_{10-11}, and one node of type 3.

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. 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.

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

	Bytecode Size	Schemina **Factor**	gzip **Factor**
libScheme	32040	23%	16%
fib	169	18%	77%
tak	582	26%	37%
earley	26271	31%	19%
conform	28599	23%	17%
mm	2550	30%	29%
destruct	3371	22%	22%
qsort	5827	57%	45%

Figure 7: Compression factors for Scheme programs.

7.2 The Scheme language

Our approach has also been applied to the Scheme language[15]. From a general stack machine called Machina our tools create a new set of instructions called Schemina.

Table 7 compares the compression factors of our technique with gzip. 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.

Table 4 presents the relative execution time of the com-

Decoder	Machine$_1$			Machine$_2$			Machine$_3$		
				Pentium					
	a	**b**	**c**	**a**	**b**	**c**	**a**	**b**	**c**
$k_r = 4, L_5, L_6$	1.81	1.76	1.52	2.19	1.64	1.48	1.38	1.07	0.96
$k_r = 5, L_6$	1.60	1.71	1.47	2.13	1.64	1.55	1.32	0.99	0.96
$k_r = 6$	1.60	1.58	1.34	2.08	1.49	1.42	1.31	0.95	0.90
				SPARC					
	a	**b**	**c**	**a**	**b**	**c**	**a**	**b**	**c**
$k_r = 4, L_5, L_6$	2.77	1.61	1.52	2.04	1.60	1.39	1.09	0.99	0.97
$k_r = 5, L_6$	2.77	1.51	1.43	2.50	1.42	1.35	1.02	0.91	0.88
$k_r = 6$	2.39	1.63	1.21	2.12	1.23	1.18	0.93	0.81	0.78
Decoder	**MachineP$_1$**			**MachineP$_2$**			**MachineP$_3$**		
				Pentium					
	a	**b**	**c**	**a**	**b**	**c**	**a**	**b**	**c**
$k_r = 4, L_5, L_6$	1.80	1.62	1.47	1.57	1.55	1.38	1.37	1.22	1.18
$k_r = 5, L_6$	1.67	1.62	1.44	1.57	1.53	1.40	1.34	1.20	1.14
$k_r = 6$	1.70	1.45	1.30	1.46	1.41	1.26	1.25	1.10	1.14
				SPARC					
	a	**b**	**c**	**a**	**b**	**c**	**a**	**b**	**c**
$k_r = 4, L_5, L_6$	2.06	1.59	1.55	1.91	1.51	1.44	1.70	1.41	1.35
$k_r = 5, L_6$	2.20	1.42	1.37	1.76	1.37	1.34	1.54	1.30	1.25
$k_r = 6$	1.90	1.29	1.16	1.60	1.27	1.16	1.46	1.19	1.14

Table 5: Relative time to execute compressed programs, based on Zipf-20, for six virtual machines, three memory access forms, and on two processors.

pressed Scheme programs. For several benchmarks there are speedups since macro-instructions increase speed and many of the new instructions have short opcodes and operands.

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 a 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 to 6 have parameters and do the same work as machines 1 to 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 $4 \cdot 10^5$ 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, if applicable, two bytes by operand. The decoding is a computed branch, indexed by the opcode, to the virtual instruction implementation. It loads its operands, emulates and jumps back to the beginning of the decoding cycle.

Table 5 presents the timing results for compressed programs, relative to the uncompressed ones. 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 compromise to make.

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.

7.4 Summary of the experiments

In conclusion, for Java, the average compression factor is around 60% for 400 classes of the JDK 1.0.2 and the ten benchmarks. For half 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

Decoding of Huffman encoded instructions has also been studied at the hardware level by several researchers [14, 17, 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.* [8] compress native code, using macro-instructions and fixing parameters, by generating a tailored VM from the intermediate form emitted by a C compiler. It is similar to Proebsting's [21] work. 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 [5] reduce program size by replacing particular repetitive sequences of instructions with a branch. The code saving is on average 5%. Cooper *et al.* [6] 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 [22] applies several techniques to compress Java class files. This work differs from ours since decompression must be performed before execution.

The work of Rayside *et al.* [23] also applies to class files, but these techniques does not apply to the bytecode itself.

Hoogerbrugge *et al.* [10] uses a similar strategy of the Thumb and MIPS16 processors [25, 13] 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 compare to the virtual program. Our work is complementary by further reducing the size of the virtual programs using compressed virtual instructions.

Lucco [18] applies compression to x86 native code using a dictionary technique to keep track of repeated short sequences of instructions. At least one decompression must be 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.* [4] compresses bytecode by replacing repetitive sequences of JVM instructions by macro-instructions. They obtain an average of 85% compression factor with a slowdown from 19% to 27%.

The work of Evans and Fraser [9] has an identical goal as ours: direct execution of compressed bytecode. Their technique avoids variable length instructions contrary to our technique. They do not report any execution times.

Debray and Evans [7] use canonical Huffman code on binary executables, but using the slow decoding technique as in sub-section 3.1. They avoid compression on frequently executed parts of the code to obtain reasonable execution speed.

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 effective, permitting a gradual compromise 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 have 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. Actually, half of the Java benchmarks have a 40% reduction in size with a negligible slowdown ($\leq 3\%$).

10. REFERENCES

[1] Java BYTEmark benchmarks: source code and results. `http://www.igd.fhg.de/~zach/benchmarks`, 1999.

[2] M. Benes, S. M. Nowick, and A. Wolfe. A fast asynchronous Huffman decoder for compressed-code embedded processors. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, Sept. 1998.

[3] J. P. Bennet and G. C. Smith. The need for reduced byte stream instruction sets. *The Computer Journal*, 32:370–373, 1989.

[4] L. R. Clausen, U. P. Schultz, C. Consel, and G. Muller. Java bytecode compression for low-end embedded systems. *ACM Transactions on Programming Languages and Systems*, 22(3):471–489, 2000.

[5] K. D. Cooper and N. McIntosh. Enhanced code compression for embedded RISC processors. In *Proc. Conf. on Programming Languages Design and Implementation*, 1999.

[6] K. D. Cooper, P. J. Schielke, and D. Subramanian. Optimizing for reduced code space using genetic algorithms. *ACM SIGPLAN Notices*, 34(7):1–9, July 1999.

[7] S. Debray and W. S. Evans. Profile-Guided code compression. In *SIGPLAN Conference on Programming Language Design and Implementation*, 2002.

[8] J. Ernst, C. W. Fraser, W. Evans, S. Lucco, and T. A. Proebsting. Code compression. In *Proc. Conf. on Programming Languages Design and Implementation*, pages 358–365, June 1997.

[9] W. S. Evans and C. W. Fraser. Bytecode compression via profiled grammar rewriting. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 148–155, 2001.

[10] J. Hoogerbrugge, L. Augusteijn, J. Trum, and R. van de Wiel. A code compression system based on pipelined interpreters. *Software - Practice and Experience*, 29(11):1005–1023, Sept. 1999.

[11] D. A. Huffman. A method for the construction of minimum redundancy codes. In *Proc. IRE*, volume 40, pages 1098–1101, Sept. 1952.

[12] T. Kemp, R. Montoye, J. Harper, J. Palmer, and D. Auerbach. A decompression core for PowerPC. *IBM Journal of Research and Development*, 42(6), Nov. 1998.

[13] K. Kissell. *MIPS16: High-density MIPS for the Embedded Market*. Silicon Graphics MIPS Group, 1997.

[14] M. Kozuch and A. Wolfe. Compression of embedded system programs. In *Proc. Int'l Conf. on Computer Design*, pages 270–277, 1994.

[15] M. Latendresse. Automatic generation of compact programs and virtual machines for Scheme. In M. Felleisen, editor, *Proceedings of the Workshop on Scheme and Functional Programming*, Sept. 2000. Available at www.iro.umontreal.ca/~latendre/publications/.

[16] M. Latendresse and M. Feeley. Fast and compact decoding of Huffman encoded virtual instructions. Technical Report DIRO-1219, University of Montreal, Nov. 2002. Available at www.iro.umontreal.ca/~latendre/publications/.

[17] H. Lekatsas and W. Wolf. Code compression for embedded systems. In *Proc. ACM/IEEE Design Automation Conference*, 1998.

[18] S. Lucco. Split-stream dictionary program compression. In *Proceedings of the ACM SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 27–34, Vancouver, British Columbia, June 18–21, 2000.

[19] A. Moffat and A. Turpin. On the implementation of minimum redundancy prefix codes. *IEEE Transactions on Communications*, 45(10):1200–1207, Oct. 1997.

[20] G. Muller, B. Moura, F. Bellard, and 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*, pages 1–20, Berkeley, June 16–20 1997. Usenix Association.

[21] T. A. Proebsting. Optimizing a ANSI C interpreter with superoperators. In *Proc. Symp. on Principles of Programming Languages*, pages 322–332, 1995.

[22] W. Pugh. Compressing Java class files. In *Proc. Conf. on Programming Languages Design and Implementation*, pages 247–258, 1999.

[23] D. Rayside, E. Mamas, and E. Hons. Compact java binaries for embedded systems. In *Cascon*, pages 1–14, Nov. 1999.

[24] E. S. Schwartz and B. Kallick. Generating a canonical prefix encoding. *Communications of the ACM*, 7(3):166–169, Mar. 1964.

[25] J. L. Turley. Thumb squeezes ARM code size. *Microprocessor Report*, 9(4), Mar. 1995.