Fast and Compact Decoding of Huffman Encoded Virtual Instructions Technical Report 1219

Mario Latendresse[1] Marc Feeley[2]

[1]FNMOC
latendre@iro.umontreal.ca
[2]Département d'informatique et recherche opérationnelle
Université de Montréal
feeley@iro.umontreal.ca

Abstract

Embedded systems often have strong 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. We use another approach, using canonical Huffman codes to generate compact custom-sized opcodes and custom-sized operand fields along with a virtual machine that directly executes the encoded operations. We present techniques that automatically generate the opcodes 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. We also discuss the relevant difficulties in generating C code for such decoders, in particular the problem of efficient program memory access. Through experiments we demonstrate the speed of these decoders. Synthetic and Java benchmarks show an execution slowdown ranging from -10% to 30%, with an average of 9% for good decoders. For the Java bytecode, the average overall compression factor is 60%.

1 Introduction

1.1 Motivation

Embedded systems are resource-constrained devices requiring careful attention to memory usage and power consumption. To serve these goals, several researchers are taking the approach of reducing program size [6, 26, 2, 13, 11, 12].

Recently, IBM has developed CodePack [11, 12] to compress programs for the PowerPC processor for the embedded market. Others have developed dual processors that switch between compressed and uncompressed modes of decoding [13, 2].

To tackle such systems in Java, Sun has taken the approach of defining a small virtual machine, the KVM[1] [26], and applying restrictions on the language and the libraries. For smart cards, this comes with major constraints, where floating-point computation, threads, and garbage collection have been removed. Tools are also provided to reduce memory usage, like Java CodeCompact^{tm}[26], which preloads class files, resolves dynamic links, and generates a complete executable ROM version. But this approach does not reduce the size of the bytecode, the target of our work.

[1]The KVM uses on the order of 128K, including libraries. The virtual machine itself is 40K-80K depending on the compiler and the target platform used.

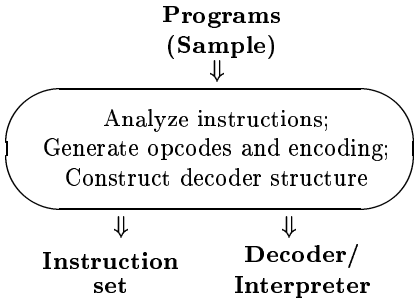


Figure 1: **Creation of instruction set, its decoder and interpreter.**

Some researchers [9, 22, 8] 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.

Some researchers [9, 4] have stated the possibility of using Huffman codes to compress programs, usually to conclude that they would, at the software level, increase decoding time to an unacceptable level. Unfortunately no clear evaluation has been done using fast software decoding techniques which this work tries to remedy.

1.2 The context of this work

We focus on the context where code decompression cannot be performed before or during 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 memory available. It is also reasonable for machines with processors much faster than memory, a trend that will increase in the future.

Figure 1 presents the general context of our work. The sample of programs, which could be as small as a single program, can be the intermediate form of compiled programs or the final form emitted by a bytecode compiler. An instruction set encoding to compactly represent the sample is automatically generated. This requires an analysis 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 same encoding as the sample. The detail techniques used to automatically generate the instruction set is partly presented in [15] and fully presented in [16]. In this paper we focus on the efficient decoding of these virtual instructions and the automatic generation of decoders.

This context allows two major applications of the work presented in this paper. The first one is the conception of virtual machines, such as the KVM, aimed at embedded systems with memory constraints. This could be done for any languages. 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 [9, 22, 8]. Further code size reduction can be obtained with a compression of the virtual instructions.

Typically, virtual instructions are "byte encoded": opcodes are encoded on a byte and operands on some byte multiple. Clearly 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 another approach for a more compact form.

Henceforth, the following setting is assumed: the opcodes are variable length canonical Huffman codes [25] generated using the static frequencies of the opcodes from a group 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.

Clearly, to gain speed, Huffman opcodes should not be decoded bit by bit; instead, blocks of k bits should be used. Such an idea has been explored previously [19, 21]. We have extended the work of Turpin and Moffat [19] to create a general algorithm capable of generating decoders with various sizes and speeds under the control of the designer of the virtual machine.

In the next section, canonical Huffman codes are presented along with a compact but slow decoding method. Section 3 presents the concepts to build much faster but slightly less compact decoders. Section 4 explains the C code's structure for all canonical decoders. Section 5 discusses the algorithm to construct the tree structure of fast and compact 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.

2 Huffman Encoding of Opcodes

We encode instruction opcodes using canonical Huffman codes [25]. These are similar to Huffman codes built by the original bottom up method of the late David Huffman [10], but the numerical value of the codes of a given length form a consecutive sequence. Such codes correspond to Huffman trees where the nodes are pushed on the same side. As it will be shown later, they have a very compact representation of the bijection between the codes and the encoded object. The average length of canonical codes are the same as Huffman codes. These opcodes are automatically generated from the frequencies of instructions from a sample of programs.

Since opcodes are canonical Huffman codes, the two terms will be used interchangeably. Moreover, the term *canonical* will often be dropped because we only use canonical Huffman codes.

Let l_c be the length of code c, $v(c)$ its value, $w \geq l_c$ a constant, and $V^w(c) = v(c)2^{w-l_c}$; in other words, $V^w(c)$ is the value of c justified in the left part of a w bits variable. The value w might simply be regarded as the width, in bits, of the processor registers. This idea of treating codes "left justified" is directly related to the manner of reading bytes from memory into a variable prior to decoding the opcodes.

Let $C = \{c_i\}$ be a set of canonical Huffman codes, $l_{\max}$ the maximum length of these codes and w a constant where $w \geq l_{\max}$. Define the vector $base^w[1 \ldots l_{\max}]$ as $base^w[i]$ is the smallest value $V^w(c)$ for all codes c such that $l_c = i$. Define the vector $disp[1 \ldots l_{\max}]$ as $disp[i]$ is the number of codes c such that $l_c < i$. Thus, the index of code c of length l_c is:

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

Therefore, C can be completely represented using space in $O(l_{\max})$. Moreover, if its length is known, its index is given by equation 1.

To show examples of opcodes, and eventually decoders for them, independently of a specific virtual machine, 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 the static frequency of instructions in actual programs. Table 2 presents some essential characteristics of the resulting Huffman canonical opcodes for $n = 200$ (the "Zipf-200" opcodes). The opcode corresponding to the probability p_i is c_i and its length is l_{c_i}. Note that the entropy[2] of the Zipf-200 probabilities is 5.9857, and that the average length of the Zipf-200 opcodes is 6.0267. Thus, the Huffman encoding is close to the optimum. For benchmarking, Zipf-20 opcodes are also used. The entropy of Zipf-20 is 3.6471 and the average length of the corresponding Huffman opcodes is 3.6689.

The vectors $base^w$ and $disp$ for Zipf-200 opcodes are shown in table 1. Note that the values of $disp$ are simply the ordinal numbers of the binary number of $base$ found in table 2.

2.1 Compact and slow decoding

Assume that the beginning of an instruction is left justified in a variable `rd`. According to equation 1, the problem of decoding the opcode c of that instruction can be reduced to finding its length. That can be done simply by a sequential search in $base$. Figure 2 shows a fragment of C code for this slow but very compact decoder based on this approach. The compressed code is in the high part of variable `rd`. Line 2

[2] The entropy is defined as $-\sum_{1 \leq i \leq n} p_i \lg p_i$, and the average length as $\sum_{1 \leq i \leq n} l_{c_i} p_i$.

```
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];
```

Figure 2: **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.**

does the sequential search. The index of the code is calculated in `crd` by line 3 using 1. Line 4 removes the opcode and line 5 does the actual branching to the virtual instruction (using `gcc`'s computed `goto`). Note that the opcode is followed by parameters or another opcode, but the search is such that the bits following the opcode can have any value.

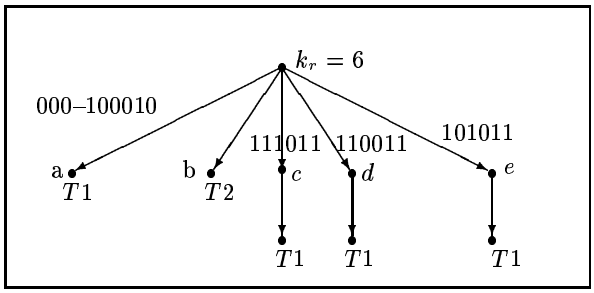
Line 2 is the major bottleneck. This is a very compact decoder since its code is small and the vectors `base_w` and `disp` only contain $l_{\max}$ elements each. For Zipf-200 on a 32 bit processor, $l_{\max} = 10$ and $w = 32$, so the two vectors use a total of 80 bytes.

3 Fast Decoding

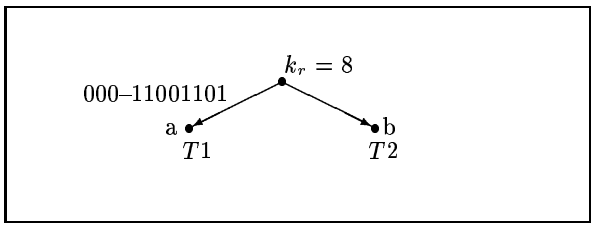
To increase speed the linear search of the length of the opcode must be avoided. This can be done with a table lookup using the index formed by the leftmost k bits of `rd`. The table contains branching addresses which either continue decoding or emulate the decoded virtual instruction. Three situations can arise:

1. The opcode is recognized by the k bits.
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 the ideal situation which occurs for all codes c with $l_c \leq k$; a direct jump to the emulation of the instruction can be done. In case 2 the length of the opcode can be used to compute its index by equation 1. Case 3 is the worst situation; the next bits must be used to continue decoding using the same technique. Therefore, 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. Type 2 nodes may share the same vector.



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 3: **Two decoder trees D_1 and D_2 for zipf-200, generated using the parameters $s_a = 4$, $s_2 = 30$, $s_3 = 25$, $t_0 = 4$, $t_2 = 10$, $t_3 = 7$. We have $k_c = 4$, $k_d = 3$ and $k_e = 2$.**

We will also use type 0 nodes. These do a linear search as in Figure 2. They are like type 3 nodes where the length is unknown but they don't consume much space since no additional vector of addresses is used. If several type 0 nodes exist they will share the same code, so that only the first one consumes space. We denote by $\overline{\nu}$ the type of node ν.

In general, interior nodes will not use the same number of k bits to do a table lookup. This makes it useful to use an algorithm to find the optimum number of bits for each type 3 node. For a node ν of type 3, we define k_ν as the number of bits used to do the table lookup. 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 and t_0 is the time of one loop iteration of line 2 in Figure 2. These timing values do not have to correspond to any real unit of time, but simply be relative to a known base value. It could be approximated by the required average number of host processor cycles.

To evaluate the space taken by the decoder, four constants are used: s_0 is the number of bytes used by the machine code for Figure 2; 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.

Figure 3 presents two decoder trees D_1 and D_2 for Zipf-200. They have been generated by the algorithm of section 5 using the specified parameters. In table 2 each opcode is shown along with the final node of decoding by the two decoders and corresponding relative time. The average time for D_1 is 19.75 and for D_2 it is 16.8; but D_2 uses almost twice the space of D_1.

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.**

```
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, transport them in 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 4: **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    += (w-nb_rd) >> 3;
nb_rd += (w-nb_rd) & ~7;
```

Figure 5: **A simple technique for line 1 of Figure 4.**

4 The Decoder C Code

Before discussing the structure of canonical decoders, it is useful to perceive them through their generated C code.

Figure 4 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 but where all of them are known to have such a length. 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 is a block of code that loads, if necessary, some additional bytes in the variable `rd`. The number of bytes loaded may vary from cycle to cycle and the exact C code to do so depend on the form of memory access that is 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. Note that it always loads a multiple of eight bits, that is the program counter points to a byte in memory, but `rd` does not necessarily contain a multiple of eight bits. Figure 5 presents an obvious portable implementation for line 1, in the case $w = 32$, but that cannot be used in practice since it is very inefficient. Section 6 presents better portable techniques.

Line 2 is the root of a decoder where the first lookup is done; line 3 does a jump to a type 2 or 3 node, or to the emulation of a virtual instruction. Note that $w - k_r$ is a constant. Similarly, 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[3].

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`. In any case, what lines 9 and 10 say, which could be done differently in some implementation, 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. This depends on the form of memory access as presented in section 6. It could return to a specific point in line 1 where it loads a specific number of bytes according to the

[3]This limits opcode lengths to no more than $w - 7$ bits.

number of bits consumed by *mne*.

5 Generating Decoder Structures

This section presents the fundamental concepts to generate an optimal canonical decoder given a set of opcodes, their dynamic frequencies, and parameters that characterize the host processor. It is based on a space and dynamic time evaluation defined as followed.

The space, in bytes, taken by a node of the decoder is defined by

$$s(\nu) = \begin{cases} s_0 + 2s_a l_{\max} & \text{if } \overline{\nu} = 0 \\ 0 & \text{if } \overline{\nu} = 1 \\ s_2 & \text{if } \overline{\nu} = 2 \\ 2^{k_\nu} s_a + s_3 & \text{if } \overline{\nu} = 3 \end{cases} \quad (2)$$

Parameters s_0, s_2, s_3, and s_a characterize the host processor as explained in Section 3.

Type 1 nodes do not use space in the decoder since they are part of the interpreter code. Type 2 nodes only consume the space of the C code since the vector of addresses `adr` is assumed completely available in the virtual machine. For type 3 nodes, the term $2^{k_\nu} s_a$ is the space taken by the vector of addresses of that node. That vector contains the code pointers to the virtual instruction implementations. The space taken by a complete decoder D is $E(D) = \sum_{\nu \in D} s(\nu)$

To evaluate the time taken by the decoder we use a vector of probabilities $P = \{p_c\}$, where p_c is the probability that opcode c is decoded while executing the program. These are not the static frequencies used to generate the opcodes, but dynamic ones. Therefore the algorithm for finding an optimum decoder is based on the dynamic frequencies of opcodes whereas their values were constructed based on the static frequencies. **This makes the algorithm to construct instruction decoders different than decoders used to decode static data.** A notable difference from the work of [19]. Moreover, decoders generated by this algorithm are in most cases faster than the ones considered in [19], since they avoid single bit decoding as much as possible. If enough space is given to construct them, all interior nodes of the decoders use several bits for table lookup, whereas in [19] single bit operations are always used after the first, and only, table lookup.

Let P_c be the path from the root to c in a decoder D. The average time taken by decoder D for the opcodes C given the probabilities P is:

$$T(C, D, P) = \sum_{c \in C} p_c \sum_{\nu \in P_c} t_{\overline{\nu}} \quad (3)$$

We now describe several sets to define precisely the structure of decoders.

Let C be a set of opcodes and $k \geq 1$ an integer. The set of opcodes in C recognized without ambiguity by their first k bits is noted R^k. The set of opcodes in $C - R^k$ for which their length is known by their first k bits is noted L^k and the set of different lengths of L^k is noted L^{k^l}. The set $C - R^k - L^k$ is noted P^k and the set of distinct prefixes of k bits of P^k is noted P^{k^p}.

A canonical tree decoder is recursively defined as either a tuple $\langle k, p, R^k, P^k, L^k, P^{k^p}, L^{k^l}, S \rangle$, $k > 0$, where the seven first values form a type 3 node and S are a (possibly empty) set of canonical tree sub-decoders. The value k is the number of bits forming the lookup index, and p is the prefix of that type 3 node. All other components have been previously defined. Note that all type 2 nodes are dispersed in the L^k components. The structure only represents explicitly type 3 nodes, since type 1 and 2 nodes can be derived from these.

Input: Opcodes $C = \{c_i\}$ (canonical Huffman codes);
Probabilities $P = \{p_c\}$ of decoding opcodes;
An upper bound B of the decoder size;
The parameters s_i of the host processor.
Output: A decoder structure $\langle k, p, R^k, P^k, L^k, P^{k^p}, L^{k^l}, S \rangle$

Let $(e, b, t, L, D) = Search((\langle \lambda, C \rangle), B, t_3, \infty, ())$
If e Then `The decoder` D `has time` t `and space` b
Else `The space` B `is insufficient to decode` C

Function $Search(Cs, b, t, t_{\text{mg}}, L)$
$Cs = \langle p, C \rangle : Cr$; $l_{\max} = \max\{l(c_i) - l_p | c_i \in C\}$
$k_{\max} = \min(l_{\max}, \lceil \lg b / s_a \rceil)$
$R_{\min} \leftarrow (\text{false}, 0, 0, (), ())$; $t_{\min} \leftarrow t_{\text{mg}}$
For k from $k_{\max}$ to 1 Do
Begin
$t' = t + t_3 \prod_{p_c \in P^k} p_c + t_2 \prod_{p_c \in L^k} p_c$
$b' = b - 2^k s_a + |L^{k^l} - L| s_2 + |P^{k^p}| s_3$
If $t' < t_{\min}$ and $b' \geq 0$ Then
$L' = L \cup L^{k^l}$
If $Cr = ()$ and $P^k = \emptyset$
Then $t_{\min} \leftarrow t'$; $S = \langle k, p, R^k, P^k, L^k, P^{k^p}, L^{k^l}, () \rangle$
$R_{\min} \leftarrow (\text{true}, b', t', L', (S))$
Else $Cs' = add(P^k, Cr)$
$(e, b'', t'', L', D) = Search(Cs', b', t', t_{\min}, L')$
If e Then $t_{\min} \leftarrow t''$;
$S = \langle k, p, R^k, P^k, L^k, P^{k^p}, L^{k^l}, D_{1..|P^{k^p}|} \rangle$
$R_{\min} \leftarrow (\text{true}, b'', t'', L', S : D_{|P^{k^p}|..})$
End
If $b' < 0$ or $t' \geq t_{\min}$ Then return $R_{\min}$
Else return (false, _, _, _)
End Search

Figure 6: **Algorithm to find optimum decoders.**

A complete decoder implementation in C can be generated from such a structure. The interpreter, including extraction of parameters, is also automatically generated given the implementation of the virtual instructions in the form of C macro-instructions. The full details can be found in [16].

The structure of an optimum compact decoder can be done by a branch and bound search as presented in Figure 6. The main input is the set of opcodes C and the dynamic probabilities p_c of decoding opcodes. The parameter B restrict the space usage of the decoder. The decoder structure found is the optimum in the sense that no other faster decoder exists, according to $T(C, D, P)$, with space no more than B.

The whole algorithm starts by calling *Search* with an empty binary prefix λ and the entire set of opcodes C, to return the decoder D, if indeed one is found. The general idea of *Search* is to start with an optimistic large k for the index lookup of the type 3 node. This has a tendency to

quickly drop the best minimum time t_{mg} found so far by the search. Once a complete structure is reached, the new minimum time is kept in $t_{\min}$ with the result in $R_{\min}$. This lower bound allows the algorithm to prune the search for a better decoder since it recursively calls *Search* only if the current time t' is less than $t_{\min}$. The variable Cs is a double ended queue, where *add* enqueue all the unidentified opcodes P^k to the not yet treated opcodes C_r. Other parts of the function precisely apply the time function T and the space function E by adjusting t and b into t' and b'. This algorithm has been used to generate all decoders of the experiments of this paper. It is practically fast enough to be included as part of the back-end compiler.

6 Accessing the Program in Memory

From the decoder tree structure as described in Section 5, C code can be generated having the general structure of Figure 4. But one important part of the C code was left unspecified, namely line 1, which loads bytes from memory into `rd`.

Getting opcodes and operands from memory can be time consuming since bit manipulations are necessary to concatenate them to the bits of `rd`. We have explored three different techniques to access memory. The first one, form-a, is obvious, 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 loaded in `rd` to allow the decoding of at least one opcode without the need for the sub-decoders to access 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 $\lfloor (w - \text{nb\_rd})/8 \rfloor$ and the number of bits to shift is $(w - \text{nb\_rd}) \bmod 8$.

This technique, as in the following form-b, loads in `rd` as many bytes as possible. The advantage is a reduced number of merging operations requiring shift and logical operations. It also allows a simpler, and faster, access to operands since they are most often hauled in before decoding the opcode. The other advantage is reduced interpreter size, since the implementation of most virtual instructions do not access memory for operands. 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 $x+1$ can be done faster than the general case.

This is possible to do, since each virtual instruction knows the number of bits to extract from `rd`, it thus 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. 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.

The advantage of this method is a slightly bigger interpreter to implement the roots, and a more complex implementation. But this second point is easily overcome with an automatic generation of decoders and interpreters as done in this work.

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 have to access memory while decoding an opcode. If it is under $l_{\max}$, as many as possible bytes from memory are merged to `rd`. For example, if $l_{\max} = 14$, $w = 32$, and `nb_rd`$= 6$, three bytes are loaded and merged to `rd`. In a way, access to memory is delayed as much as possible.

The advantage of this method is a reduced number of merging operations to `rd`. This shows up in the experimental results.

The disadvantage is that we can no longer suppose, at the entry of the implementation of a virtual instruction, that there are between $w-7$ and w bits in `rd`. 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 since there are $w-7$ bits in `rd` after decoding the opcode (assuming the implicit $l_{\max} < w-7$). Furthermore, the code of the interpreter is larger since more virtual instructions as to implement its own access to memory.

7 Experimental Results

For the experimental results in a realistic setting we use a Java virtual machine applied on ten benchmarks [1] and the entire JDK 1.0.2 library. We use a set of synthetic benchmarks to demonstrate the worst scenarios. We also succinctly report the results of our approach applied to the Scheme language.

For all experiments two processors are used: a Pentium II running Linux and a Sparc Ultra-1 running SunOS 5.6 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

To demonstrate our approach in a realistic setting we use the Harissa's implementation [20] of the Java Virtual Machine (JVM): most virtual instructions' implementation are

Benchmark	**Absolute Time Uncompressed**		**Relative Time Compressed**				**Compression Factor of JVM Code**	**Size of JVM code in bytes**
	Pentium	SPARC	Pentium		SPARC			
			$C_{k_r=7}$	$C_{k_r=10}$	$C_{k_r=7}$	$C_{k_r=10}$		
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.**

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
MP_1	MP_2	MP_3	MP_1	MP_2	MP_3
0.40	0.49	0.85	2.32	2.83	3.76

Table 4: **Absolute time, in seconds, to execute uncompressed programs, based on Zipf-20.**

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.**

unchanged but branching instructions must be modified to branch on non-byte boundaries. Harissa implementation 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 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 opcodes are Huffman encoded using frequencies of instructions in over four hundred classes of `classes.zip` from JDK 1.0.2. So, these opcodes accomodate a large number of JVM classes. The 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 instruction JVM machine. This extension was done automatically by our tool to generate virtual instruction sets from a sample of programs [15, 16]. 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 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 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 must be kept in registers to have good performance. The Pentium code reveals that not enough registers are available to do that.

The worst speed results are the Fourier and Bitfieldops benchmarks. It 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 does not access object fields as frequently as the other benchmarks. Since the `getfield` and `putfield` 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 overall results show the practicality of the approach where even half the benchmarks have a 40% reduction in size with a negligible slowdown.

7.2 Synthetic benchmarks

The Java benchmarks demonstrate the applicability of the approach in a realistic setting. On the other hand, for three important aspects, we think that synthetic benchmarks can answer some questions without the complexity of the JVM instruction set. First, the JVM implementation is fairly complex and raises the question of hidden overhead by the emulation of the virtual instructions. Second, the statistical distribution of instructions in the benchmarks and the JDK libraries might be skewed favorably towards our approach. Finally, the extraction of parameters has an impact on the speed of interpretation, so that the distribution of operand lengths is an important factor to calibrate. Therefore, we also present synthetic benchmark timings, where the frequency of instructions, their granularity, and their operand lengths are precisely specified.

For the synthetic benchmarks, we use six virtual machines of different granularity. These different granularities allow 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 using zipf-20 probabilities.

In the first machine, all twenty virtual instructions add one to one of twenty counters (c_i); in the second machine each instruction does two additional dummy integer operations; in the third one, each instruction does two additional memory accesses to simulate a dummy stack. Clearly, the granularity of the instructions are fairly small in machine 1 and it increases slowly upto machine 3. 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. For all machines, the first instruction stops the execution when counter c_1 exceeds $4 \cdot 10^5$. We use the same program for the six machines: it is a sequence of the twenty instructions, from instruction 1 to 20, performing a loop of $4 \cdot 10^5$ iterations; that is the last instruction does a jump to the first instruction. These programs are compressed using the opcodes generated by the zipf-20 probabilities. Three decoders are applied on all six machines executed on two host processors.

Another interpreter was used to decode the uncompressed form of the same programs. In these cases the programs are bytecoded, using one byte for an opcode and, if applicable, two bytes for an operand. The decoding is done by performing a computed branch to the virtual instruction using the value of the opcode as an index on a vector of adresses. The virtual instruction loads its operands, does the emulation and jumps to the beginning of the decoding cycle.

Table 4 presents the absolute time in seconds of the execution of the uncompressed programs. Table 5 presents the timing results of executing the compressed programs, relative to the uncompressed ones. First, we can see that the memory access form is important. If only form-a were used, the results would be disappointing. Even in the case of $k_r = 6$, which completely decodes all opcodes in one step, the slowdown is at least 60% for all machines on all processors. The two other forms are clearly superior. Form-c is very often faster on Pentium and SPARC for the machines without parameters; but on some cases it is not. We conclude that the two forms are close enough in performance to keep them both available for the designer of the virtual machine. And since form-b generates more compact interpreters there is an informed compromise to make.

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.

In conclusion, from this experiment, we can conclude that even with virtual instructions doing almost no work, as in machine 1, the decoding itself is around a 25% overhead (as in form-c used with a $k_r = 6$ decoder). This is an extreme artificial setting to openly show the worst scenario. On the other hand, if we have instructions with no parameter and enough granularity, a speed up can be observed. This experiment also shows that the form of access is important, since the obvious form-a is clearly slower and should not be used in practice. The form-b and form-c provide a compromise between speed and space, where form-c is faster but taking up more space.

7.3 Scheme language

Further experimental results for the Scheme language have been reported in [15]. These results also show that decoding of compressed virtual instructions is not a substantial overhead for well tailored Scheme systems. It also shows that our general technique creates an encoding clearly more compact, by a factor of two in some cases, than a hand crafted virtual instruction set for Scheme.

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 with a slowdown of less than 1% for every 1% decrease in size. For half the benchmarks, the slowdown is hardly noticeable with a decrease in size of 40%. This shows the high practicality of the approach. The synthetic benchmarks shows the general expectation of graceful degradation of speed with decreasing decoder size. The Scheme results demonstrate that although the method of compression is very general, it can achieve as good results as a well conceived design of a bytecoded virtual machine tended solely for the compactness of one language.

8 Related Work

Decoding of Huffman encoded instructions has also been studied at the hardware level by several researchers [14, 17, 3]. 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 [22] 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 [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 [23] 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.* [24] also applies to class files, but these techniques does not apply to the bytecode itself.

Hoogerbrugge *et al.* [9] uses a similar strategy of the Thumb and MIPS16 processors [27, 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.

Raeder *et al.* [5] compresses bytecode by replacing repetitive sequences of JVM instructions by macro-instructions but without further compression.

The compression of bytecodes for virtual machines was addressed by Wilner for the SDL language on Burroughs B1700 [28]. It uses Huffman codes but all decoding was done at the microcode level, bit by bit. This is much too slow at the software level.

To our knowledge, no previous work has been published analyzing fast Huffman decoders, at the software level, of compressed virtual instructions.

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 as 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 constraint.

An algorithm to automatically generate such decoders becomes indispensable if the instruction set is automatically constructed. We have shown an efficient algorithm to construct compact optimum decoders given a memory constraint.

Huffman decoding is not the only major difficulty for quickly interpreting compressed virtual instructions. The access to memory is also very important. Two 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 9% to 27% depending on the processor and the size of the decoder used with an average reduction in size of 40%. Actually, for half the benchmarks there is no noticeable slowdown with a reduction in size of 40%. This has been applied to more than 400 classes of the JDK 1.0.2 library. This shows the practicality of the approach.

To our knowledge, no previous work has studied direct interpretation, at the software level, of compressed virtual

instructions, where opcodes are Huffman encoded.

Acknowledgments

Thanks to Keith Cooper for helpful comments on a draft of this paper.

References

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

[2] ARM. *An Introduction to Thumb.* Advanced RISC Machines Ltd., March 1995.

[3] Martin Benes, Steven M. Nowick, and Andrew 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.

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

[5] Lars Raeder Clausen, Ulrik Pagh Schultz, Charles Consel, and Gilles Muller. Java bytecode compression for embedded systems. Technical Report RR-3578, Inria, Institut National de Recherche en Informatique et en Automatique, 1998.

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

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

[8] Jens Ernst, Christopher W. Fraser, William Evans, Steven Lucco, and Todd A. Proebsting. Code compression. In *Proc. Conf. on Programming Languages Design and Implementation*, pages 358–365, June 1997.

[9] Jan Hoogerbrugge, Lex Augusteijn, Jeroen Trum, and Rik van de Wiel. A code compression system based on pipelined interpreters. *Software - Practice and Experience*, 29(11):1005–1023, September 1999.

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

[11] IBM. *CodePack: PowerPC Code Compression Utility User's Manual. Version 3.0.* International Business Machines (IBM) Corporation, 1998.

[12] T.M. Kemp, R.M. Montoye, J.D. Harper, J.D. Palmer, and D.J. Auerbach. A decompression core for PowerPC. *IBM Journal of Research and Development*, 42(6), November 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] Mario Latendresse. Automatic generation of compact programs and virtual machines for Scheme. In Matthias Felleisen, editor, *Proceedings of the Workshop on Scheme and Functional Programming*, Rice Technical Report 00-368, September 2000.

[16] Mario Latendresse. *Génération de machines virtuelles pour l'exécution de programmes compressés.* PhD thesis, Université de Montréal, May 2000.

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

[18] Steven 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] Alistair Moffat and Andrew Turpin. On the implementation of minimum redundancy prefix codes. *IEEE Transactions on Communications*, 45(10):1200–1207, October 1997.

[20] Gilles Muller, Bárbara Moura, Fabrice Bellard, and Charles 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] Yakov Nekritch. Decoding of canonical huffman codes with look-up tables. Technical Report 85209-CS, University of Bonn, Department of Computer Science, January 2000. Wed, 06 Sep 2000 09:44:01 GMT.

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

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

[24] Derek Rayside, Evan Mamas, and Erik Hons. Compact java binaries for embedded systems. In *Cascon*, pages 1–14, November 1999.

[25] Eugene S. Schwartz and Bruce Kallick. Generating a canonical prefix encoding. *Communications of the ACM*, 7(3):166–169, March 1964.

[26] Sun. *Java 2 Platform Micro Edition (J2ME) Technology for Creating Mobile Devices.* Sun Microsystems, May 2000.

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

[28] W. T. Wilner. Burroughs B1700 memory utilization. *AFIPS FJCC*, 41:579–586, 1972.