Billy get your guns: fast barrel-shift decoding for in-place execution of Huffman-encoded bytecode streams

Technical Report TR-0001

Hwangho Kim[1], Kirak Hong[1], Sungho Kim[1], Taekhoon Kim[1],
Jiin Park[1], Yousun Ko[1], Bernd Burgstaller[1] and Bernhard Scholz[2]
Yonsei University[1], The University of Sydney[2]

Abstract

The use of ubiquitous computing devices in memory-constrained environments increases the need for techniques that reduce the memory footprint of applications while leaving them in an executable form. Heterogeneous ubiquitous computing platforms require applications to support a variety of CPU architectures. Virtual Machines (VMs) provide hardware abstraction and a code representation with high compression potential. Efficient interpreters are thus ideal for heterogenous memory-constrained ubiquitous computing platforms. We apply Huffman compression to both opcodes and operands of bytecode. Our VM is capable of executing compressed bytecode without prior decompression. The slow-downs due to on-the-fly decoding of compressed bytecode are less than a factor of 7. Compressed bytecode is up to 72% smaller than binary code, and up to 87% smaller than a "plain" bytecode representation.

1. Introduction

Embedded systems have severe resource constraints which resulted in research for the reduction of memory footprints using code compression. CodePack is a code compression system for binary files used by IBM with their PowerPC processor architecture [5]. The ARM Thumb extension is another approach to reduce code size of program binaries [9]. Binary code has limited compression potential compared to stack-based VMs: with binary code, all operands of an instruction must be specified, whereas with a stack-based VM many operand references are implicit references to the VM's evaluation stack. Bytecode thus has a higher compression potential than binary code, which makes VMs attractive for systems where program memory is at a premium. With VMs, instructions are typically byte-encoded, i.e., opcodes and operands are encoded in units of bytes. This encoding trades space for speed: fixed lengths for opcodes and byte-alignment of opcodes and operands facilitate fast decoding at the cost of a large program code size. To reduce the memory footprint of bytecode, Latendresse et al. [6] employ Huffman coding to the opcodes of a bytecode application. Instead of decompressing the compressed program prior to execution, opcodes are decoded on-the-fly.

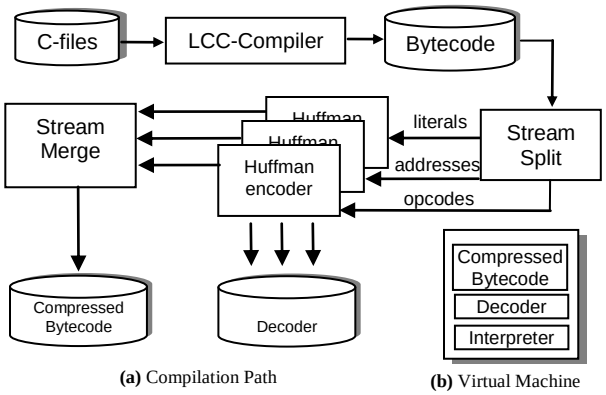

Figure 1. Compilation Path and VM

In this paper we extend the work of Latendresse et al. to employ Huffman encoding to both opcodes and operands (i.e., opcodes, literals and addresses of bytecode instructions) to obtain higher compression rates. We implemented our approach with TinyVM, a VM originally proposed in [3]. TinyVM is based on vmgen [4], an interpreter generator for VMs. Vmgen takes instruction descriptions of a VM as input and produces an interpreter in C. Vmgen employs advanced interpreter techniques such as threaded code (representing an instruction as a pointer to the interpreter routine that implements the instruction), TOS-caching (keeping the topmost VM stack element in a register) and superinstructions (combining frequently occurring instruction sequences into a single instruction). The approach in [3] uses *off-line decompression*, i.e., compressed bytecode is expanded in a memory buffer and converted to threaded code; threaded code is then executed by the VM. It was observed in [4, Table 3] that threaded code allows very fast bytecode interpretation, but consumes four times as much memory as machine code. The memory buffer for decompression further increases memory consumption, which is not

viable for memory-constrained embedded systems. To circumvent construction of threaded code at run-time, we devise a decoder that allows us to execute compressed bytecode on-the-fly. To speed up decoding, a processor register is dedicated to cache part of the instruction stream and to decode from there (see, e.g., [6]). Traditionally this processor register is called a "barrel". With our decoding scheme we devise optimizations for barrel-based decoding of Huffman-encoded bytecode streams. The contributions of this paper are

- Huffman-encoding of multiple streams (i.e., opcodes and operands) to reduce memory footprint,
- support for efficient decoding of compressed bytecode without prior decompression,
- integration of Huffman-decoding with the vmgen VM generator,
- fast barrel-shifting schemes for Huffman decoders,
- an implementation of our approach with TinyVM, a VM for C,
- experimental evidence for the validity of our approach. From this experiment, slow-downs due to on-the-fly decoding of compressed bytecode are less than a factor of 7. Compressed bytecode is up to 72% smaller than binary code, and up to 87% smaller than a "plain" bytecode representation.

This paper is organized as follows: Section 2 describes our approach to Huffman encoding/decoding of bytecode. Section 3 describes the integration of our approach with the vmgen VM generator. In Section 4 we present experimental results. In Section 5 we draw our conclusions and discuss future work.

2. Huffman-encoded bytecode streams

Figure 1 (a) shows the compilation path to generate compressed bytecode images. C source files are compiled to bytecode using our bytecode backend for the LCC C compiler [7]. Bytecode images are split into three streams, for opcodes, literals and addresses. This separation yields a smaller number and consequently shorter codewords in each category, which is crucial for obtaining high compression rates. After encoding, the three streams are merged into a single stream and stored as an assembly file. Merging is done according to the original structure of bytecode instructions (i.e., opcodes are again interleaved with the corresponding operands). An alternative representation would be to store opcode, literal and address streams in different sections. However, this would require three separate instruction pointers. For hardware architectures with a small number of registers, dedicating two additional registers for instruction pointers largely increases register pressure and decreases interpreter performance.

2.1 The General Encoder/Decoder Mechanism

Integration of the Huffman decoder into the instruction fetch mechanism of the VM [6] requires precomputed decoder tables. The compressed bytecode is a bitstream, with the instruction pointer pointing into this bitstream. Because Huffman codewords are not byte-aligned, the VM instruction pointer requires bit-granularity.

At run-time, the decoder reads the next k-bits from the stream and moves the instruction pointer k-bits to the right. The k-bits are used to decode the next codeword from the stream. To speed up decoding, a decoder table is used. Codewords can become very long (up to 21 bits in our applications). Using a single table is not viable in such a setting, because the table would be too big. In fact, the prefix property [16] of Huffman codes (no codeword can be a prefix of another codeword) would result in many unused table entries. To overcome this issue, the decoder table is split into subtables, which slows down decoding. The sizes of the subtables again depend on a speed/space tradeoff.

The decoder mechanism consists of several code fragments that are executed via the decoder table. The general layout for our decoders is as follows.

Table 1. Decoding scheme

```
decode0:
    val = get_nbits(<k>);
    goto *decoder_table<nr>[val];
    ....
code<y>:
  shift_left(<k>); return y;
```

To decode a new codeword, the decoder starts in `decode_0`. It reads k-bits from the stream and uses the resulting number as an index into the first decoder subtable. The subtable contains the location of either a valid codeword (which will then be returned to the interpreter), or more bits need to be processed from the stream to decode the codeword (i.e., another `decode<x>` fragment is executed). If a codeword is identified, more bits might have been read than actually necessary. In this case the bitstream pointer needs to be rectified by calling procedure `shift_left()`. Procedure `shift_left()` will be omitted if the bitstream pointer does not need to be adjusted.

To generate decoder tables, we have a simple algorithm that starts with the computation of a canonical Huffman code. (A Huffman code is canonical if the numerical values of the codewords of a given length form a consecutive sequence [16]). Starting from the root of the canonical Huffman tree, we generate the subtables for the decoder. A waste occurs if we have identical entries in a subtable, i.e., with less bits the code words could be identified. We increase the size of a table until our waste threshold is saturated. We substantially simplified the decoder scheme from [6]

which uses a complicated branch-and-bound scheme to compute decoders for canonical Huffman codes.

As far as the authors are aware of, we have the first instruction format for bytecode that compresses literals and addresses, splits bytecode instructions into different streams, and employs inplace-decoding of multiple compressed bytecode streams. With our encoding scheme, we had to make decisions for encoding

- literals that are operands (integers, floats aso),
- target addresses of conditional and unconditional jumps,
- addresses of global variables, and
- target addresses of call statements.

We use alternating codes, i.e., different Huffman codes for opcodes, literals and addresses. Depending on the opcode, the decoder of the VM decides which decoder(s) are to be used to decode the operands of instructions. As an example, consider the instruction "`ADDRG_OFFS foobar 10`": after decoding the opcode, the VM uses the decoder for addresses and the decoder for literals to decode operand `foobar` and the literal `10` from the instruction stream.

Encoding target addresses of conditional and unconditional jumps turned out to be a hard problem. We decided to use instruction-pointer-relative addresses of fixed bitlengths, because such code is position-independent (for ubiquitous computing devices that do not have memory management units, position-independent code is an advantage if memory needs to be reorganized, e.g., because of code updates). The reason why we did not choose Huffman codes for jump targets is that Huffman codes do not have a monotonicity property, i.e., if opcode x is smaller (i.e., uses less bits) than opcode y, it is not necessarily true that the codeword for opcode x is smaller than the codeword for opcode y. Finding the distances between jump targets is a non-trivial problem, because in between there might be other jump instructions whose length depend on the current jump instructions. This results in a simultaneous equation system that is very hard to solve. For this reason we decided to encode the relative addresses of jumps as signed 16 bit distance values.

2.2 Decoder Optimizations

Decoding Huffman codes requires many fine-grained (i.e., bit-level) memory accesses. Function get_nbits() introduced in the previous section reads a small number of bits each time a lookup in a decoding table becomes necessary. Reading a few bits each time from memory would be too inefficient; a processor register is therefore dedicated to cache the part of the bitstream ahead of the VM's instruction pointer. This processor register is called "barrel". Because function get_nbits repeatedly requests a certain number of bits from the bitstream, managing the barrel effectively is an issue. Note that because of our bytecode representation as a single bitstream with opcodes and operands interleaved, the decoders for opcodes, literals and addresses can share one barrel.

With the approach of Latendresse et al. [6], the decoder barrel gets reloaded at the beginning of the decoding routine. Extending this approach to three streams, the barrel will get reloaded at the beginning of the opcode, literal and address decoder. Reloading the barrel from the bitstream in memory is a costly operation. We introduce three schemes, namely D1, D2 and D3, for efficient barrel reloading of multiple-stream Huffman decoders.

D1 decoder: As shown in Figure 2, the decoder barrel is reloaded byte-wise. This requires 4 memory accesses and 4 shift operations for a 32-bit barrel (nb_rd is the number of bits still valid from a previous barrel load). On architectures that do not require aligned memory accesses, we can load multiple bytes at a time, as depicted in Figure 3. This approach saves memory accesses, but on Little Endian architectures the byte order will be reversed. On the Intel x86, the CPU instruction bswap reverses the byte order of a register value. The code in Figure 3 uses an inline assembly statement to correct the byteorder (Lines 2 - 5. The corrected data from the bitstream is then appended to the barrel (Line 6).

```
#define Byte(x)   *(byte_pos + x)

barrel |= (BYTE(0) << 24 | BYTE(1) << 16
           | BYTE(2) << 8 | BYTE(3)) >> nb_br;
```

Figure 2. Byte-wise instruction fetch

```
1  barrel_tmp = (*(unsigned long *)byte_pos);
2  asm("bswap %0"
3      : "=r" (barrel_tmp)
4      : "r"  (barrel_tmp)
5  );
6  barrel |= (barrel_tmp>> nb_rd);
```

Figure 3. D3 fetching: Reloading 32 bits at once after correction of endianness

D2 decoder: The reason to fill the barrel at the beginning of the decoder is to ensure that enough bits are in the barrel to decode the next codeword. Surveying our application bytecodes, we notice that space is distributed unevenly between opcodes, literals and addresses. A typical ratio would be opcodes 27%, literals 48% and addresses 25%. Interestingly, the high space consumption of literals was due to a large number of literals from a small pool that in turn resulted in small Huffman codewords. With addresses on the other hand, we observed many different address values which in turn resulted in longer codewords. So we adjusted the Huffman table generator to fill the barrel at only the first stage of the opcode and address decoder. As a result, the number of barrel reload operations will be reduced, which improves decoder performance.

D3 decoder: Because our decoders for opcodes, literals and addresses share one decoder barrel, we can optimize barrel reloading among decoders. If the bytecode of an application contains no instruction where the length of the codewords for opcode plus operands is larger than the width of the decoder barrel, it is sufficient to reload the barrel at the beginning of the opcode decoder. A barrel under-run may occur with some applications. Surprisingly, from all surveyed applications only Gzip showed a barrel under-run. For the ongoing shift to 64-bit architectures, substantially bigger applications will become possible with this decoding scheme.

3. Integration with Vmgen

TinyVM has been implemented entirely with the vmgen VM generator [4], and we expect this method to become frequently used with future generations of VMs. Currently vmgen only supports threaded code, which has a very low code density. For memory-constrained devices, it is thus essential to integrate in-place execution of compressed bytecode with vmgen.

Vmgen provides a well-defined interface to a VM implementor. This interface includes (1) wrapper functions that allow the VM to access data on the VM evaluation stack or from the bytecode instruction stream, (2) the VM instruction specifications itself, (3) different threading schemes for efficient bytecode instruction dispatch. To see how this interface works, consider the instruction specification of TinyVM's cnst_i4 instruction.

```
cnst_i4 ( #iss_l -- iss_l )
```

This instruction takes one value of type iss_l from the bytecode stream and pushes it onto the VM stack. Type iss_l is a type definition that is part of the TinyVM interpreter specification; it denotes C long values. The details of vmgen instruction specifications can be found in [4]. Pushing the value 22 would be expressed as follows in bytecode:

```
cnst_i4 22
```

Vmgen generates the implementation depicted in Table 2 for the const_i4 instruction. In Line 3 vmgen uses the vm_Cell2iss_l macro to retrieve a value of type iss_l from the instruction stream and stores it in variable iss_l declared in Line 2 (IMM_ARG(IPTOS) refers to the current position of the instruction stream pointer). To make this work, the vmgen user has to provide a macro definition for all used data types.

Table 2. Instruction implementation for cnst_i4

```
1 LABEL(cnst_i4) /* cnst_i4 ( #iss_l -- iss_l ) */ {
2 long iss_l;
3 vm_Cell2iss_l(IMM_ARG(IPTOS),iss_l);
4 sp += -1;
5 vm_iss_l2Cell(iss_l,spTOS);
6 NEXT_P2;
7 }
```

For TinyVM, two of those macros are given in Table 3. As can be seen from the vm_Cell2iss_l macro definition, function decode_numeric() is called. This function is provided by our decoder to decode literals. Likewise for addresses: macro vm_Cell2iss_target is used by vmgen to retrieve a Huffman-encoded address value from the instruction stream. In this macro function decode_symbol() is called, which is again provided by our decoder to decode address values. The function definitions shown in Table 3 are generated by our Huffman encoder at the time the application bytecode is encoded (see Figure 1). These functions contain the decoding tables described in Section 3.

Table 3. Macros to invoke address and literal decoders

```
#define
  vm_Cell2iss_l(_cell,_x) {
       _x=(signed long)decode_numeric();}

#define
  vm_Cell2iss_target(_cell, _x) {_x=(void *)decode_symbol();}
```

To dispatch bytecode instructions, vmgen allows the user to specify a threading scheme for instructions. A threading scheme consists of macros that have to be provided by the vmgen user.

Table 4. Vmgen threading scheme for Huffman-encoded bytecode

```
# define IP (byte_pos)
# define SET_IP(p)
   {SET_ABS_IP ((unsigned long)p);}
# define NEXT_P2
   {  goto *labels[decode_opcode()]; }
```

Vmgen requires macros to read the instruction pointer (IP()), set the instruction pointer (SET_IP) and dispatch to the next bytecode instruction (NEXT_P2). The threading scheme for Huffman-encoded bytecode is depicted in Table 4. Variable byte_pos is the pointer of our barrel into the instruction stream. SET_ABS_IP sets the instruction stream pointer. Finally, NEXT_P2 dispatches to the next bytecode instruction by calling the decoder's decode_opcode() function and looking up the address of the instruction in the VM's lookup table. Table 2 shows how macro NEXT_P2 is used as part of the instruction implementation of the bytecode instruction cnst_i4.

Although vmgen was not designed to be used with Huffman-decoders, its clear-cut interface allows us to perform integration of our Huffman decoder as part of the generation process of the decoder tables. As depicted in Figure 1 (b), a TinyVM virtual execution environment consists of (1) the compressed bytecode image, (2) the decoder consisting of decoder tables plus the vmgen interface, and (3) the interpreter that is generated by the TinyVM VM specification. All three constituents are generated automatically; our infrastructure

for Huffman-encoded bytecode can be easily integrated with other VMs specified with the vmgen VM generator.

4. Experimental results

We implemented our Huffman code infrastructure for in-place execution of compressed bytecode with TinyVM, a VM for C. We conducted extensive experiments to show the validity of our approach. All experiments were conducted on an Intel Xeon 1.866GHz dual core processor running Linux. We used the following benchmarks for evaluation. From the MiBench [13] benchmark suite: FFT, JPEG_enc, JPEG_dec, CRC32, BasicMath, Dijkstra, StringSearch and adpcm. From the Spec2k [14] benchmark suite: Gzip, Bzip and MCF.

We investigated the following issues: (1) we compared the code size among plain bytecode, Huffman encoded bytecode (three-streams) and binary code compiled with the x86 backend of the LCC compiler, (2) we determined the performance overhead caused by in-place execution of Huffman-encoded bytecode vs. plain bytecode, (3) we compared the compression rates possible with opcode encoding only (Latendresse's approach) to our approach were opcodes and operands are compressed, and (4) we determined the performance improvements of our barrel load mechanisms D1, D2 and D3 from Section 2.

Table 5 compares the code sizes obtained from plain bytecode, binary code and Huffman-encoded bytecode. Compression ratios r were computed according to the following formula:

r = (uncompr_size - compr_size) / uncompr_size

Compared to plain bytecode, Huffman-encoded bytecode saves between 82.79% and 87.1% memory space (column "vs. Plain"). Huffman-encoded bytecode is between 47.7% and 72% smaller than binary code (column "vs. Binary").

Figure 4 shows the performance overhead caused by in-place execution of compressed bytecode vs. plain bytecode. Slowdown factors vary from 3.5 for StringSearch to 6.9 for Bzip. These results are in-line with the results obtained in [17]: Latendresse et al. encoded opcodes only and observed slow-downs up to a factor of 2.5 in their synthetic benchmarks. A direct comparison is however difficult, because Latendresse et al. benchmarked Java and Scheme VMs, whereas TinyVM is a VM for C.

Figure 5 compares the compression ratios between Latendresse's results and 3-stream Huffman compression vs. plain bytecode. For Latendresse, the size of their operands is not specified in their paper. We computed a lower bound 'Latendresse-B' and an upper bound 'Latendresse-W' for the code size achievable with Huffman-encoding of opcodes only. It should be mentioned that the best case is hardly achievable, because it requires dedicated VM instruction for all possible operand sizes, which will unduly increase the size of the interpreter. As depicted in Figure 5, encoding of opcodes, literals and addresses ("three streams") yields a compression rate that is up to 40% higher than opcode-only encoding.

Figure 6 compares the execution times for the barrel reload mechanisms. As already mentioned, 'Gzip' did not work with the D3 scheme because of a barrel underflow. All other benchmarks worked with the D3 scheme. Column "old-decoder" denotes the decoder without the D1-D3 optimizations. If the codeword sizes of an application allow it, D3 is the most efficient barrel reloads mechanism. Otherwise D1 and/or D2 should be used.

Table 5. Code sizes and compression ratios for bytecode and binary code

Benchmark		Plain in TinyVM	Binary in LCC	Huffman three-streams in TinyVM	vs Binary	vs Plain
		Size (bytes)			Compression ratio (%)	
MiBench	FFT	8532	2467	1290	47.71	84.88
	JPEG_enc	509936	140728	73234	47.97	85.64
	JPEG_dec	409508	135192	70516	47.85	82.79
	Stringsrarch	6278	2808	998	64.46	84.11
	adpcm	4208	1067	543	49.11	87.10
	CRC32	1592	528	217	58.90	86.37
	BasicMath	7524	3224	1029	68.08	86.32
	Dijkstra	3114	1480	405	72.63	84.32
Spec2k	Gzip	140348	42412	21553	49.18	84.64
	Bzip	88144	28093	13415	52.24	84.78
	MCF	37988	10325	5119	50.42	86.99

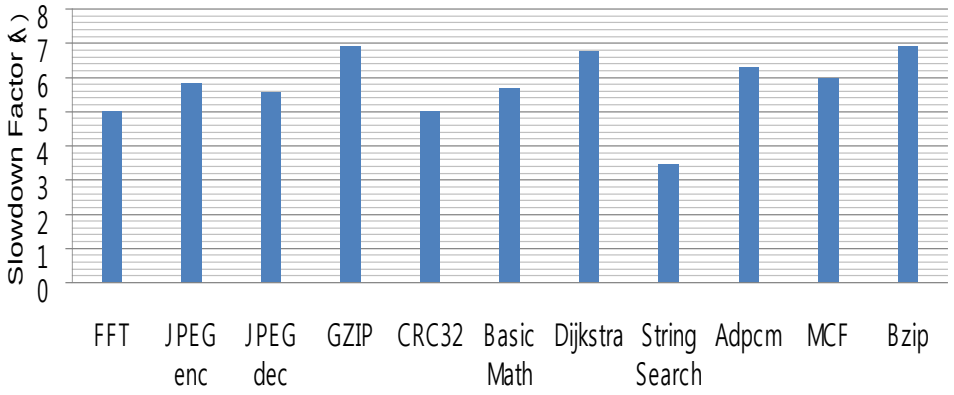

Figure 4. Slowdown factor of three-stream codes on plain bytecode

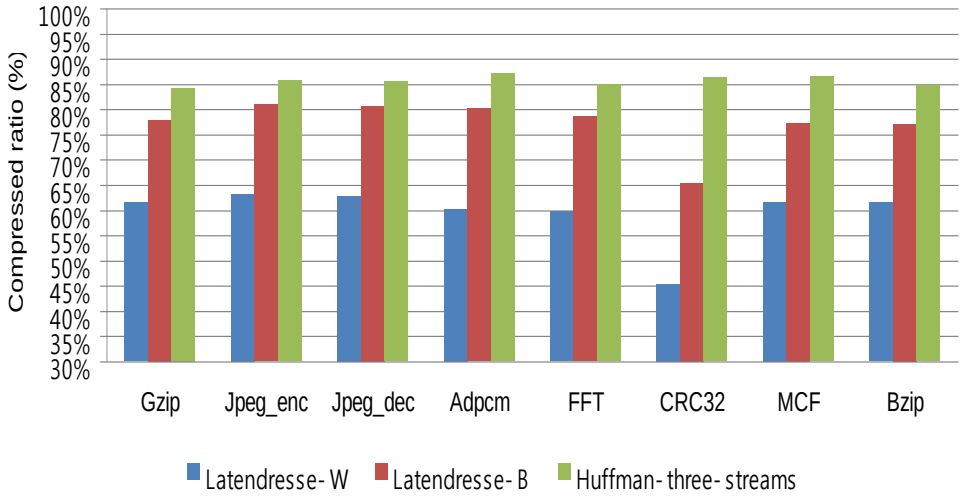

Figure 5. Compression rates of Huffman-compressed opcode-streams and three-streams on plain bytecode

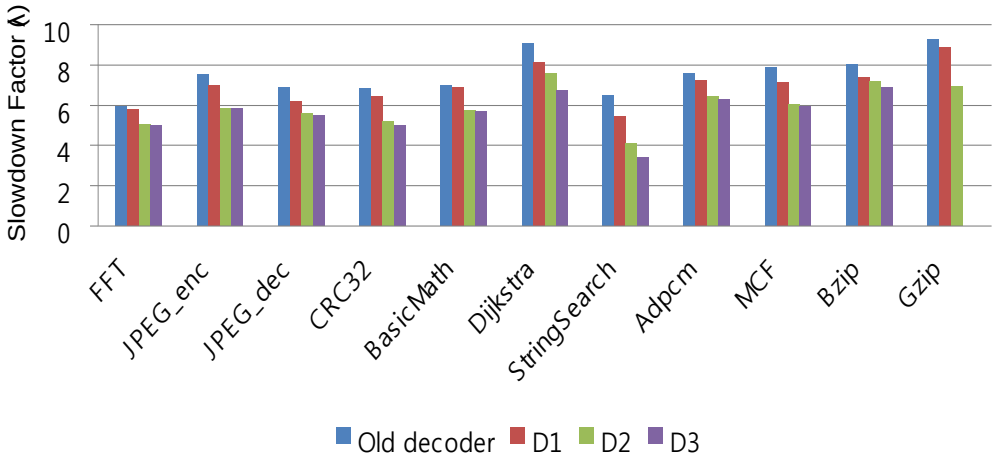

Figure 6. Slowdown of the basic decoder ("old") and three-stream decoders vs. plain bytecode

5. Conclusion

We have presented a technique for in-place execution of Huffman-compressed bytecode. Unlike previous approaches, we compress opcodes and operands to achieve higher compression rates. We introduced several techniques that increase the efficiency of the decoder. We implemented our approach with a VM for C and conducted extensive experiments with MiBench

and Spec2k benchmark programs to show the validity of our approach. Observed slowdown factors for in-place execution of compressed bytecode vary between a factor of 3.5 and a factor of 7. Compared to plain bytecode, the usual code representation of VMs, Huffman-encoded bytecode saves between 82.79% and 87.1% memory space. Huffman-compressed bytecode is between 47.7% and 72% smaller than binary code, which makes it a viable code representation for memory-constrained ubiquitous computing platforms. The hardware abstraction provided by VMs is an additional advantage with our approach. As for future work, we plan to investigate VM superinstructions and second order codes.

7. References

[1] D. A. Huffman. "A Method for the Construction of Minimum-Redundancy Codes", In Proc. IRE, volume 40, September 1952, pp. 1098 – 1101.

[2] T. Egan, "Practical Data Compression Methods for Embedded Systems", Sun Nuclear Corporation 425-A Pineda Court, Melbourne, FL 32940

[3] B. Burgstaller et al., "An Embedded Systems Programming Environment for C" In Proc. of the Euro-Par Conference, Springer LNCS, 2006.

[4] Ertl, M.A. et al., "vmgen - A Generator of Efficient Virtual Machine Interpreters", Software - Practice and Experience 32, 2002. pp. 265 -294

[5] IBM. CodePack : "PowerPC Code Compression Utility User's Manual", Version 3.0., 1998.

[6] M. Latendresse et al., "Fast and Compact Decoding of Huffman Encoded Virtual Instructions", Technical Report DIRO-1219, University of Montreal, November, 2002.

[7] Hanson, D.R. et al., "A Retargetable C Compiler: Design and Implementation", Addison Wesley, 1995.

[8] Debray, S. et al., "Profile-Guided Code Compression" Proc. PLDI'02, ACM Press, pp. 95~105.

[9] J.L. Turley. "Thumb squeezes ARM code size", Micro processor Report, 9(4), March 1995.

[10] Sun Microsystems, "J2ME Building blocks for Mobile Devices", CLDC, May 19, 2000.

[11] Bell, J.R., "Threaded Code", CACM 16, 1973.

[12] IEEE Standard Testability Method for Embedded Core-Based Integrated Circuits, 1500-2005, 2005 [Online]. Available: http://www.ieee.org

[13] Guthaus, M.R. et al., "MiBench: A free, commercially representative embedded benchmark suite", Proc. of WWC'01, 2001.

[14] Standard Performance Evaluation Corporation: Spec CPU 2000, 2000, http://www.spec.org.

[15] T. Lindholm et al., "The Java Virtual Machine Specification", 2nd edition.

[16] I. Pu, "Fundamental Data Compression", Elsevier, 2006.

[17] M. Latendresse et al.,, "Generation of fast interpreters for Huffman compressed bytecode", Science of Computer Programming 57 (2005) 295 – 317.