Proc. Estonian Acad. Sci. Eng., 2005, **11**, 4, 269–285
https://doi.org/10.3176/eng.2005.4.02

Binary code compression based on decision trees

Tamás Gergely, Ferenc Havasi and Tibor Gyimóthy

Department of Software Engineering, University of Szeged, Árpád tér 2, 6720 Szeged, Hungary; {gertom, havasi, gyimothy}@inf.u-szeged.hu

Received 4 August 2005, in revised form 30 September 2005

Abstract. More and more embedded systems are used in the world. These machines have limited resources as, e.g., the background storage size. It is important to increase the storage capacity of these machines. A possible solution for this is compressing the files before they are written on the storage device. This is usually done by a general compresssor. There are files containing text or binary data and in many cases binary program code. A general compressor compresses all of them with almost the same efficiency. But a compressor, specialized for one type of input, would compress files of that type much better; thus using many special compressors would save space and increase the virtual capacity of the storage device. Using this idea, our goal was to create a compression algorithm for binary program code. Most compression methods can be separated into two parts: the model and the coder. In this paper we introduce a decision-tree based modelling method. We combined this method with an arithmetic coder and applied it in a modified JFFS2 file system of a Linux distribution, running on a PDA machine. The original file system used only one method for file compression (zlib), the modified file system uses many compressors, including our model-based one. This PDA machine has an ARM processor; thus our method was implemented for the compression of ARM instructions. The results were very promising: depending on the parameters of the method, at least 12.6% of the 13 MB image, created with only zlib compression, were saved, and it costed boot speed slowdown only at most 3 times.

Key words: embedded system, code compression, decision trees.

1. INTRODUCTION

There are more and more embedded systems spreading all around the world (e.g. mobile phones, PDAs). The capacity of these small computers is limited in many ways. These limitations include processor speed, available energy, dynamic memory and background storage size.

In many cases the background storage is some kind of static memory (e.g. flash), which keeps its state without consuming energy. The size of this memory is minimized by the embedded tools. But on the other hand, embedded systems contain more and more programs and data. To save space, a good solution is to compress programs and data.

This compression is usually done by a compressed file system. Such a file system logically stores fixed-sized blocks, but physically stores them compressed in variable-length blocks, logically increasing the capacity of the storage device. In most cases the used compression algorithm is a general one, which can compress programs and data with the same efficiency (e.g. gzip).

The blocks to be compressed can be classified. There are blocks containing text data, binary data or program code. File systems compress all these blocks with the same compression method. However, special compression methods could be used to compress different kinds of blocks. The size of a block, compressed by its special compressor, is usually smaller than the size of the same block, compressed by the general compressor. Thus compressing each block with its special compressor makes the compressed block smaller and saves space.

To take advantage of this, we created a compression algorithm for the binary program code. We aimed this algorithm on the ARM instruction set, which is the most frequent code set in embedded systems. The implementation is called ARMlib. However, the method can be applied for other binary codes that satisfy some requirements (e.g. well separatable instruction/parameter/condition bits).

There are lots of code compression methods introduced in the literature [1]. These methods usually can be separated into a model and a coder part. Our method also has these parts: based on previous works of Fraser [2] and Garofalakis et al. [3], we used special decision trees as the model and an arithmetic coder [4] as the coder.

The method was implemented and tested on real iPAQ machines with a specially modified file system. The original JFFS2 [5] file system was designed for flash devices and included zlib compression. This was modified to handle both ARMlib and zlib compressed blocks at the same time [6]. This file system modification itself is not a subject of this paper, but it was necessary for trying our compression method in a real environment.

The compression efficiency of JFFS2 was improved with ARMlib: on an image of about 25 MB uncompressed size, from 1.7 to 2.6 MB were saved, using the algorithm. This means that the compressed image was from 12.6 to 19.4% smaller when ARMlib was used, compared to the original (zlib-only) compressed image; thus this amount of flash memory could have been saved.

The drawback of ARMlib is its speed. The boot time of the iPAQ, when ARMlib was used, was from 2 to 4 times the original boot time (when only zlib was used). However, except some extreme cases, the slowness of ARMlib is usually not recognized by a human user after the boot procedure has been finished.

In this paper we introduce a decision tree based code compression method, which combines Fraser's idea of using decision trees as compression models [2]

and the effective tree construction algorithm of Garofalakis et al. [3], which were originally designed for building trees for classification. We applied this method in a real environment [6]. Namely, the JFFS2 file system in the Familiar Linux distribution was modified to use both zlib and our method on an iPAQ machine. (The method is usable on other binary machine codes too. We tried it for Thumb where the method saved 3.5 to 8.5% of the uncompressed size, which means saving 9 to 20% relative to the zlib-compressed size. However, these ratios are for pure thumb code compression and not for system-wide use due to lack of a Thumb hardware unit.)

The rest of the paper is organized as follows. In Section 2 some terms of compression and decision trees are introduced and in Section 3 some existing code compression and decision tree building methods are described. In Section 4 our method is introduced. In Section 5 an existing implementation on iPAQ machines is introduced and in Section 6 the results are presented. In the last section the results are summarized and some issues of further improvement are mentioned.

2. BACKGROUND

In this section we briefly describe what compression means and how it usually works, then we shortly describe decision trees.

2.1. Compression

The term "compression" can be defined as [1]: "storing data in a format that requires less space than usual". In other words, compression means representing data in a form that is smaller than the original representation. The term "decompression" means the inverse of compression, thus restoring the data in the original form.

2.1.1. *Theoretical background*

The theory behind compression is based on the results of *information theory*. In this section we will review some terms of information theory. We define the input of a compression method as a sequence of input symbols. These symbols can be the bits or bytes of the input as well as more complex entities. These entities are usually called tokens. The input sequence may contain values from a fixed set of symbols (token values). The basic idea of most compression algorithms is to assign a *code* to each symbol in such a way that the sequence of the codes will be shorter than the sequence of the symbols.

Each symbol in the input has a *probability*. By giving a shorter code to a more frequent symbol and a longer code to a less frequent one, the overall size of the output sequence will be smaller than the size of the input sequence. Most compression methods use this idea to produce smaller output sequences.

Formally that can be expressed as follows. Let $A = \{a_1, a_2, ..., a_N\}$ be a set of symbols. Let $X = x_1, x_2, ..., x_m$ be a sequence with $x_i \in A, (i = 1, ..., m)$. Now, $a \in A$ has a probability $P_X(a) \geq 0$ in the sequence X, with $\sum_{a \in A} P_X(a) = 1$.

We can compute the *information content* of X. The less information is stored in X, the shorter encoded sequence can represent it. The information content can be measured by the entropy of X using the following formula:

$$H_A(X) = -\sum_{a \in A} P_X(a) \log_2 P_X(a).$$

This formula gives the minimum average number of bits required to encode one symbol in the sequence X. If $H_A(X)$ is multiplied by m, we get the theoretically minimal size (in bits) of the encoded sequence of X.

2.1.2. *Compression model*

As mentioned above, most of the compression methods are based on the information content of the input sequence by assigning shorter codes to more frequent tokens. Thus the compression method can be separated into two parts: 1) gathering information about the input sequence and 2) assigning suitable codes to the tokens. The first component is called *modeller*, the second component is called *coder* and it uses the model, provided by the modeller during the code assigning process. Most compression methods contain separately a modeller and a coder, although these are usually fine-tuned on each other. Many modellers can be used with the same coder and vice versa; thus these two can be treated as separate research topics.

The goal of the model is to provide a good probability distribution on the next token in the input: for all token values tell what is the chance of the event that the next token has the said value. Modelling can mean almost anything: the simplest model is the probability distribution of the token values in the input, but the probability distribution provided by the model may change from token to token. For example, the values provided by the model may depend on the value of the last token.

The coders also vary. There are coders that directly assign codeword to each individual input token value (e.g. Huffman) and there are coders that assign a final codeword to a sequence of tokens (e.g. arithmetic). The *decoder* implements the inverse of the coder (it restores the tokens from the codes); however, it uses the same model that was used by the coder.

2.2. Code compression

Code compression covers the compression of almost any form of a program, including intermediate representation or binary program code but excluding source code (which is sooner a special case of text compression than code compression).

Although all these forms are considered as “code”, there can be many differences between them; e.g. the IR code can be some kind of a tree (e.g. Abstract Syntax Tree) while binary code is a sequence of machine instruction.

2.3. Decision trees

Decision trees are used for storing information about a set of objects. More precisely, decision trees are trees that provide some information about the target-attribute of an object using some other attributes of it. A usual application of them is the classification of objects, where the object is put into one of the given classes based on some properties (attributes) of the object [7,8].

In Figure 1, the objects are given by their attributes and assigned to a class ($+$ or $-$). The decision tree on this figure encodes this classification.

The tree contains an expression with attributes (*predictors*) in each of its internal nodes. Such an expression can be the attribute itself (as in the example) or the comparsion of the attribute to one of its possible values or more complicated expressions too. An outgoing edge is assigned to each of the possible results of the expression in the node. These edges end in subtrees. In the leaves of the decision tree the information (target-attribute) is stored that corresponds to the decisions made on the route from the root to the actual leaf.

To extract the information about an object (which is given by its attribute values) from a decision tree, first evaluate the decision in the root using the attribute values of the given object, then check the end of the edge, assigned with the result. If there is a subtree then repeat the process on that. When a leaf is reached, it will contain the required information.

Decision trees are mainly used because they can be automatically built. Tree construction algorithms require only a great number of examples (where the target-attributes are also known) to automatically create a decision tree. The best known decision tree building algorithms are ID3 [7] and C4.5 [8].

ATT#1	ATT#2	ATT#3	CLASS
1	A	x	+
2	A	x	-
3	B	y	+
3	C	y	+
2	B	x	-
3	B	x	-
1	A	y	+
1	C	y	-
2	A	y	-
1	B	y	-

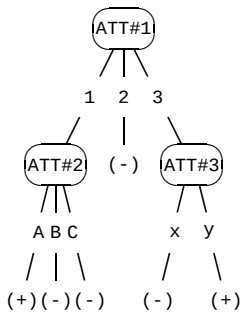

Fig. 1. A simple decision tree.

3. PREVIOUS WORKS

In this section previous works on binary code compression and decision tree building are described, showing their advantages and disadvantages.

3.1. Code compression methods

Code compression is used for various reasons. The goal can be the reduction of energy consumption, which can be the result of the smaller stored size and/or of the smaller data size, sent through the channels between hardware elements. Sometimes compresson has no other reason than saving space, thus improving the capacity of the storage device. For any reasons, code compression methods can be very different.

We concentrate on binary code compression although we borrow some concepts from other code compression as well.

Benini et al. created a *transparent* compression method [9], where the compressed code is decoded by a hardware unit between the memory and the CPU. The decoder unit is a simple table that contains 255 most frequent codes, assigned to a byte-long codeword. The 256th value is an escape character to state an uncoded instruction. The most frequent codes come from statistics and the codewords are assigned using Minimal Hamming Distance.

Wolfe et al. described a *block-based* compression method, usable on hardware with memory cache [10]. The cache pages are stored in the memory in compressed form and decompressed into the cache when a cache miss occurs. The relation between the in-cache and real memory addresses are resolved using a Line Allocation Table (LAT). Many coders were tried, and a version of Huffman coding was proposed by the authors for application.

Breternitz and Smith enhanced the previous method [11] and eliminated the LAT. The memory addresses in the code are modified to contain a cache page address and an offset of the target instruction. This solution created more problems, for example, when the program runs through a cache page boundary without a jump. (This was later solved by automatically generated jump instructions.) The best compression ratio, achieved in [11], was 0.56.

Laketsas et al. [12] improved Wolfe's method. They proposed the decomposition of RISC instructions into *streams*, thus different parts of the instructions (e.g. operation code, target register) are encoded in different sequences. They also proposed arithmetic coding. Their models were Markov models and dictionaries. Their average compression ratios were 0.5 to 0.7.

The work of Lefturgy et al. [13] is similar to Wolfe's solution, but they assign codewords not only to single instructions but to instruction sequences too. The used model is a dictionary, the codewords have fixed size. The CodePack method of IBM [14] is similar to this but much more complicated. The 32-bit long instructions are divided into two 16-bit long parts and these are encoded with a variable-length coding. The decoder is also a hardware unit, but the software-based decoding was also studied. The compression ratios for these methods were about 0.6.

The methods above are designed for hardware decompression. Their main task was to reduce the energy consumption of the embedded systems, thus the simplicity was more important than the greater compression ratio. Their models are usually based on some kind of a dictionary. In our case the energy consumption is secondary, the decompression can be done by software and compression ratio is more important. Thus our model can be more complex.

In [2] Fraser creates a model for code compression using machine learning methods. The coder and decoder are not the subject of his paper; he focused on finding the relevant statistical information existing in the code to be compressed and automatically extracting it.

Fraser works on an intermediate representation (IR), not on the binary code. The information he extracts from this representation are probability distributions usable by his coder. He stores these distributions in the "leaves" of this decision tree like model. To do this, predictors are required that describe the context of the actual token. Fraser used the last 10 to 20 token values before the actual token (so-called "Markov" predictors) as predictors, and some computed predictors like the stack depth. He used a simple tree-building algorithm to infer his model automatically: it gets a large number of internal representation codes and builds the tree for this data set. Fraser reduced the size of the model making a DAG (*directed acyclic graph*) from the tree by merging similar leaves. The presented results showed a compression ratio of 0.19 on IR code. But these results did not count the size of the model, which can be very large.

3.2. Decision tree building

Decision trees can be automatically generated on large and representative *training data sets* and thus it is easy to use them. One of the best known tree building algorithms is ID3 [7]. It is based on entropy gain. Let X be a set of objects, T the target-attribute and A a non-target attribute. Let $\mathbf{T}$ and $\mathbf{A}$ denote the set of target and non-target attribute values. Now define the probability $P_X(t)$ and set $X_{A,a}$ for all $t \in \mathbf{T}$ and $a \in \mathbf{A}$ as

$$P_X(t) = \frac{|\{x|x \in X, T(x) = t\}|}{|X|},$$

$$X_{A,a} = \{x|x \in X, A(x) = a\}.$$

Now the entropy gain of attribute A on the set of objects X, using the formula $H_S(X)$ introduced in Section 2.1, is

$$EG_A(X) = |X| H_{\mathbf{T}}(X) - \sum_{a \in \mathbf{A}} (|X_{A,a}| \, H_{\mathbf{T}} (X_{A,a})).$$

Tree building works as follows: a set of objects (X), whose attributes are known, is assigned to the root; the *best attribute* (A), which produces the highest entropy gain ($EG_A(X)$) is then selected; the set of objects is split into subsets, based on the best attribute ($X_{A,a}$) and each subset is assigned to a child of the root.

The algorithm is then invoked for the children. If all attributes produce negative entropy gain, the node becomes a leaf, encoding information of the target attribute values of the objects.

A problem can be the *overfitting*. If a tree, trained on a noisy training set (that contains many errors), exactly fits the training set, then the answers the tree gives will also contain errors. Two methods can solve this problem.

The first is applied during tree building. When a new node is examined to decide whether it is to be expanded or not, some *stopping criterions* are checked. These may depend on various properties of the node and on the assigned training set. If any of these criteria becomes true, the node is not expanded.

The other method is applied on the fully built tree. The subtrees are examined and replaced with leaves, if necessary. This is called *pruning*. It gives more accurate result than the previous method, because all subtrees are well known during pruning, while in the previous method the subtrees are not known before the stopping decision. On the other hand, it may happen that a great subtree is built first and then dropped during pruning.

An enhancement of the ID3 algorithm is C4.5 [8]. It contains a pruning algorithm that works on a rule-set derived from the tree. Each leaf is represented by the decisions, made in the tree from the root to that leaf, then these rules are merged and sorted and the resulted rule list is used.

In [3] Garofalakis et al. introduced some methods for efficent building of decision trees. Their trees are built for classifications and used the MDL (Minimal Description Length) measure for pruning.

Their pruning method works on the tree. Let R be the root of the (sub)tree to be pruned, C_{leaf} is the MDL cost of a leaf that encodes the information of the training set assigned with R. The C_{node} cost of the (sub)tree can be recursively computed. The pruning algorithm works as follows:

Procedure Prune(R)
  C_{leaf} := cost of a leaf at node R
  $C_{\mathrm{node}} := \Sigma_{i \in \mathrm{Children}(R)}$ Prune(R_i) +
    cost of encoding internal node R
  If $C_{\mathrm{leaf}} \leq C_{\mathrm{node}}$ Then
    Replace the subtree rooted at R with a leaf
    Return C_{leaf}
  Else
    Return C_{node}

In [3] two enhanced versions of this methods were described. In the first, the size of the tree (number of nodes in it) can be maximized by replacing subtrees of the pruned tree with leaves, using the dynamic programming method. In the second, the minimal precision of the tree can be set. This uses the previous method, initially setting the limit on the tree size to 1, then increasing the limit by 1 until the precision of the tree reaches the required value.

In the same paper a third method was also described that can use the limits set for tree size during tree building phase (using *branch and bound* technique).

4. ARMlib CODE COMPRESSION FOR ARM

In this section our compression method is described through *ARMlib*.

ARMlib is a function library that contains functions for ARM binary code compression. ARM binary code was selected because ARM instruction set [15] is the most popular instruction set in embedded systems.

Based on the work of Fraser [2], our model in ARMlib is a decision tree, whose leaves contain probability distributions. For an application of ARMlib, the decision tree is pre-built (*trained*) on a host machine using a great number of binary code blocks (*training set*).

The building of the decision tree is based on the papers [2,3]. The building is automatic: it optimizes the image size; thus not only the compression ratio is minimized (because it would result in a very precise but huge model) but the size of the tree itself is also considered. These models provide best performance (i.e. greatest compression) on code sequences that are very similar to the code they were trained on. By setting the parameters of the training, the efficiency and speed of the compression can be varied.

An arithmetic coder, based on [4], is used as the coder in ARMlib.

4.1. The parts of ARMlib

ARMlib consists of 3 main parts: model generator, compressor and decompressor modules. The model generator creates the model from a great number of example programs (the training data set). The compressor is to compress and the decompressor is to decompress the binary ARM code; both use the previously created model. The tokenizer and detokenizer algorithms are also parts of the ARMlib. Tokenizer transforms the raw ARM binary code into the sequence of tokens usable by the modeller and coder, and detokenizer transforms the sequence of tokens into ARM binary codes after decompression. Figure 2 shows how the parts of the ARMlib work together.

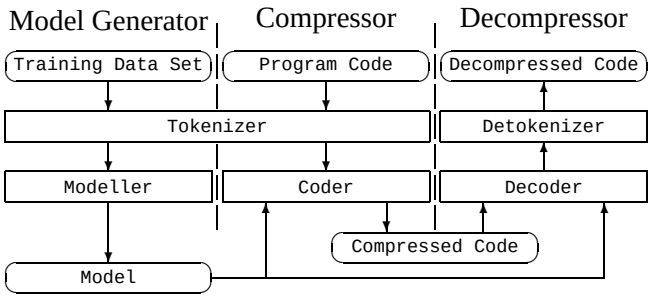

Fig. 2. The structure of ARMlib.

4.2. Pre- and post-processing

The input of model generation and compression and the output of decompression are in the ARM binary code. But the presented methods work on a sequence of tokens, so we need to transform the raw 32-bit long ARM codes into tokens and the tokens back to ARM codes. These transformations are done by the tokenizer/detokenizer modules.

ARM instructions are 32 bits long that would be too large for our method, so an instruction must be split into more smaller tokens. In ARMlib the instructions are fragmented into 8 tokens which are 4 bits long each. So the first step of the model generation and the compression is to create a sequence of tokens from the sequence of ARM instructions, which is done by the tokenizer module. The last step of the decompression is to create the ARM instruction sequence from the token sequence. This is done by the detokenizer.

The 4-bit long tokens were chosen for two reasons: the first is that the ARM instructions functionally consist of parts whose length in bits is a multiple of 4. The second reason is that 4 bits means 16 token values and this small number is very good for the model. The order of the tokens in the token stream of an instruction was determined and fixed after some preliminary measurements had been made on test data. This was necessary because the value of token T_a of an instruction may imply some values of token T_b of the same instruction, but T_b has no such influence on T_a. In this case T_a must precede T_b.

The method can be easily applied to any architecture whose instructions have a fixed length. And if the bits of the instructions can be assigned to functional groups, the method might be as successful as in case of ARM.

4.3. The model

ARMlib uses special decision trees. The decision trees of ARMlib have the following traits: objects are the tokens; predictors are the last 16 tokens before the actual token and of the type of the actual token (e. g. destination register); the information, stored in the leaves, consists of the probability distributions of the possible token values.

In [2] Fraser uses many kinds of *computed* and *reduced* predictors. In our case, predictors that required complex computations did not improve the efficiency of the method so much as to be worth of being used. The "last n token" type predictors were combined with only one computed predictor, which states the type of the actual token. The last 8×2 tokens were also used as predictors, thus the last 2 values of each of the 8 types of tokens were stored as predictors. These are easy to compute and the possibility of decomposing the input into many streams still remains; however, the tree-building algorithm has to decide if it is worth using the type predictor at the root of the tree or not. (Using it corresponds to the stream separation.)

In the internal nodes of the tree many types of decisions can be used. The simplest decision is a predictor itself (as in ID3 and C4.5 trees). A node with this type of decision has so many children as many values the predictor has (8 in case of the type predictor, 16 otherwise). Another type of decision is the comparison of a predictor with one of its possible values. This can be an equality or a less-than comparison. A node with this kind of decision has two children: one for the *true* and one for the *false* value.

The MDL measure, used in [3], is not enough to build an optimal tree in this case, because the size of the tree and the size of the compressed code have to be minimized together. To exactly compute this value, three things must be known: 1) the storage method and size of the tree and its subtrees, 2) the code to be compressed and 3) the coding method and its compression effect on the code. If these three things were known, the optimal tree could be built. However, certain problems may arise.

The way the tree is stored is known and the size of the tree or any of its subtrees can be exactly computed as long as the tree itself is not compressed. But it may happen that the tree itself is also compressed in some way (which is not a crazy idea in a compressed file system). In this case the compressed sizes of the subtrees depend on each other and cannot be computed individually. The code to be compressed is known. Each subtree will compress those tokens that are in the training set of one of the subtree leaves. The coding method is also known, but to compute the exact size of the compressed code, the coding or at least a fake coding (that computes only length information) must be done. This means too much computations, thus we have to make a deal.

From the uncompressed size of the subtree (which is exactly computed) the size of a subtree and a pre-defined *compression-multiplier* are computed. This means a few additional computations and approximates the exact final size well (if the given multiplier is good). The size of the tokens, compressed by a subtree, is approximated with the sum of the entropies of the training sets, associated with the leaves of the subtree. This is also a good approximation if the output of the coder is close to the entropy.

Thus the building of an ARMlib model for a given token sequence works as follows. A training set is made from the token sequence, which is not more than assigning the predictor value to each token. The tree is built for this training set using the general tree building algorithm. When this is done, the pruning algorithm is invoked for the root with the previously described cost computations. Some kind of parallelization of building and pruning is implemented in ARMlib. As soon as a subtree is built, the pruning method is invoked for its root immediately. This may reduce the maximal tree size in the memory during the tree building.

4.4. The coder

ARMlib uses arithmetic coder [4]. The concept of the arithmetic coder is that it does not assign codewords, but assigns an interval between 0 and 1 to the tokens.

It also assigns an interval to the sequence of tokens (this interval is computed from the intervals of the tokens). The codeword, assigned to a sequence of tokens, is a random number in the final interval, stored with a precision high enough for correct decoding.

Arithmetic coder produces a codeword, whose length is very close to the entropy. Technically the halving of the size of the actual interval means one more bit in the codeword. This allows the arithmetic coder to virtually assign a fraction of a bit to a token, whose probability is greater than 50% (while Huffman coder assigns at least 1 bit to each token regardless its probability).

5. AN ARMlib IMPLEMENTATION

ARMlib was tested in a real environment, on *iPAQ* machines [16]. These machines have *StrongARM* processors, 32 MB of memory and 16 MB of flash. The operating system used was *Familiar Linux* with a modified *JFFS2* (Journaling Flash File System 2) file system.

The original JFFS2 file system stores the files logically in fixed size blocks. But it is able to compress these blocks in a transparent way when they are stored in the flash, which means that the user of the file system does not see the compression. In this way the capacity of the flash is logically increased. The used compression methods are the methods of zlib (Fig. 3a).

The compression methods of ARMlib were integrated into JFFS2 as can be seen in Fig. 3b. The two compressors work side by side in the file system. Each block of the file system stores the information in the form it was encoded; thus decompression can be done by the appropriate decompressor.

The model generation is done on a host machine. First the files and the directory structure of iPAQ are created. Then the files (programs) that contain the ARM code are selected, but some of them that have to be compressed by zlib for some

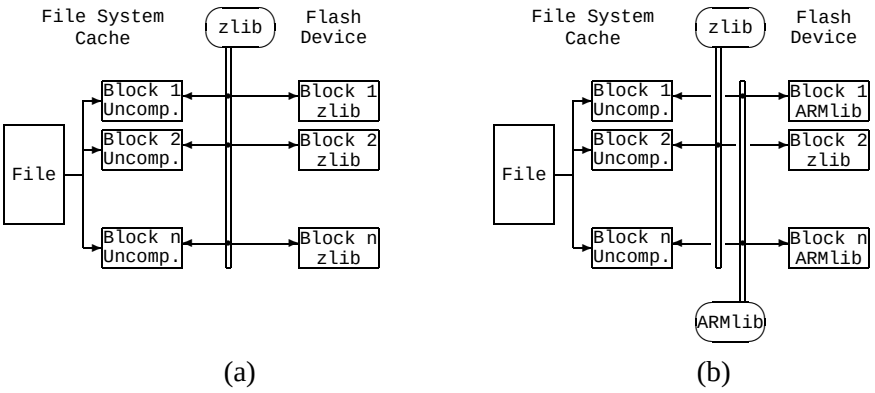

Fig. 3. JFFS2: (a) original; (b) modified.

technical issues (e.g. kernel files, ARMlib itself) are excluded from the selection. The selected programs are split into blocks (as in the file system), and the blocks that do not contain ARM code are also dropped. The training set is created from the remained blocks. It is possible to classify the blocks by creating many distinct training sets and building a separate tree for each such set.

After the model generation has been finished, the models are added to the directory structure and the binary image is made. During this all blocks are compressed with both ARMlib and zlib, and the smaller variant is kept. The block stores the identifier of the model it was compressed with, if necessary (if there were more than one model). Model file blocks can only be compressed with zlib.

After the image is ready, it is uploaded into the iPAQ. The bootloader (which knows only zlib) loads the kernel modules, including ARMlib. At initialization the file system reserves some memory. The models, listed in a configuration file, are loaded into this area and the rest of the reserved memory becomes the file cache. After this ARMlib is ready to work.

6. RESULTS

Measurements were done in two environments. In the test environment the modeller, coder and decoder were programs like gzip, and all were run on a host machine. The second environment was the JFFS2. Here the modeller and the coder ran on a host machine, but coder and decoder functions are used on the iPAQ. This environment was used to test the practical applicability of the method. The uncompressed size of the image of the iPAQ was 25 MB.

In both cases the compression was block-based. We used 4-KB long blocks. Only those blocks were used for training that contained the ARM binary code.

6.1. Compressed size

As can be seen in Fig. 4, the larger the input, the better the compression ratio is. The results were done in the test environment by training a tree for each test object and compressing its blocks with the tree trained on it. The test objects contained 1 to 12 blocks. (In real environment we did not make such measurement, but on a smaller image with about 18 MB uncompressed size ARMlib produced about one percent worse results.)

Different image sizes can be seen in Fig. 5. The size reductions relative to the image, compressed with zlib only, are between 12.6 and 19.3%. Trees with binary nodes only are better than trees with multi-value decisions (because multi-value nodes can be substituted with binary nodes, but the contrary is not true). A small image size enlargement is observed when one model was replaced by 7 smaller models (these were trained on 7 randomly separated distinct parts of the original training set). The results showed that one tree is more effective in compression size (at least when the split of the data set is random) than the 7 smaller models together, but the (de)compression speed is better for 7 models.

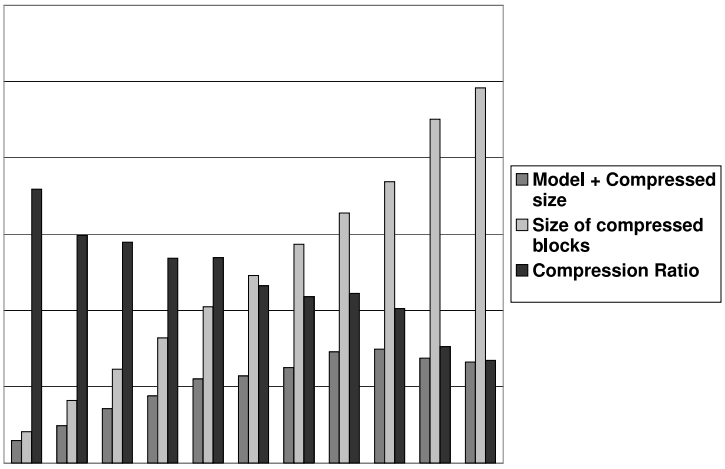

Fig. 4. Compressed size of different inputs.

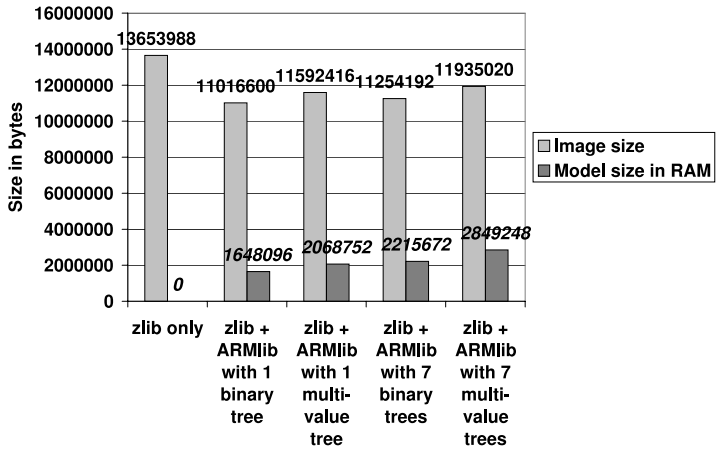

Fig. 5. Compressed sizes.

The same figure shows the sizes of the models in the memory. Binary models are smaller, and the 7 smaller models together require more memory.

6.2. Speed of the compression

The speed of the compression is worse than the gzip speed. Compression and decompression in ARMlib requires almost the same amount of time, but this makes decompression very slow relative to zlib (in zlib, decompression is 10 times faster than compression). Fortunately, Linux caches the file system. This means that not all accesses to the block provoke compression or decompression (usually only the

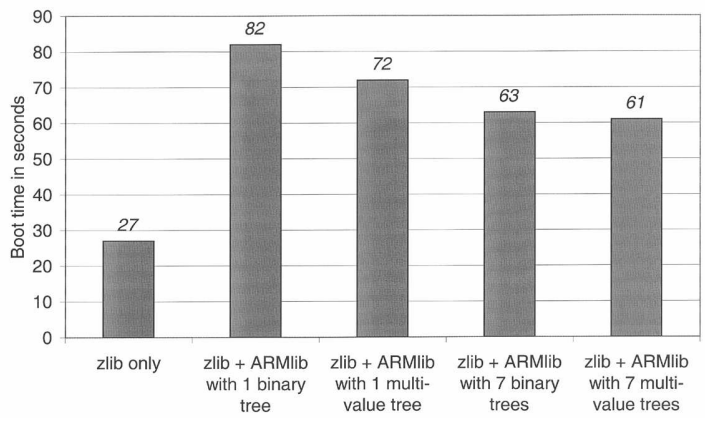

Fig. 6. Boot times.

first and the last accesses). This improves the average speed of the file system and makes ARMlib slowness almost undetectable to a regular user.

The boot time (from turn-on to the set-up of the system) is 2.3 to 3 times of the original boot time (Fig. 6), but considering the saving in size (12.6 to 19.3%) this seems to be a good deal.

7. CONCLUSION AND FUTURE WORK

Our goal was to create a code-compression method which compresses ARM binary codes better than current algorithms do. We combined a method that used decision trees as compression models [2] and an efficient tree-building algorithm [3]. In addition, the tree-building and pruning phases were combined within the method. The method is aimed for the ARM instruction set, but it can be applied for many instruction sets that satisfy some requirements.

The implementations of the functions of the method were collected in a library ARMlib. ARMlib uses an arithmetic coder [4]. ARMlib was used in the JFFS2 file system [5] on iPAQ machines. This modified file system [6] used either zlib or ARMlib compression on a block. The use of ARMlib reduced the image size by 12.6 to 19.3% depending on the parameters of tree building. The method can be used in situations when the size is more important than speed.

Possible improvement can be the automatic sorting of the input. The implemented method was tested with one tree built on the full image, and with 7 trees built on 7 separated parts of the image. But these parts were randomly separated. A good classification of the image blocks would improve not only the performance but the compression ratio of ARMlib too.

The measurement functions can also be modified to estimate the speed of the compression (perhaps by utilizing the fact that compression speed is proportional to

the depth of the tree). The compression ratio and compression speed can be varied by changing a parameter.

REFERENCES

1. Beszédes, Á., Ferenc, R., Gyimóthy, T., Dolenc, A. and Karsisto, K. Survey of code-size reduction methods. *ACM Comput. Surv.*, 2003, **35**, 223–267.
2. Fraser, C. W. Automatic inference of models for statistical code compression. In *Proc. Conference on Programming Language Design and Implementation*. Atlanta, 1999, 242–246.
3. Garofalakis, M., Hyun, D., Rastogi, R. and Shim, K. Efficient algorithms for constructing decision trees with constraints. Bell Laboratories, 2000; http://www.bell-labs.com/user/rastogi/sigkdd2000.ps
4. Witten, I. H., Neal, R. M. and Cleary, J. G. Arithmetic coding for data compression. *Commun. ACM*, 1987, **30**, 520–540.
5. Woodhause, D. JFFS: The Journaling Flash File System. In *Ottawa Linux Symposium*, 2001; http://www.linuxsymposium.org/2001/proceedings/pdf/jffs2.pdf
6. Havasi, F. Increasing compression performance of block based file systems. JFFS2 Improvement Project – BBC. http://www.inf.u-szeged.hu/jffs2/bbc.php
7. Quinlan, J. R. Induction of decision trees. *Mach. Learn.*, 1986, **1**, 81–106.
8. Quinlan, J. R. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, San Mateo, 1993.
9. Benini, L., Macii, A., Macii, E. and Poncio, M. Selective instruction compression for memory energy reduction in embedded systems. In *Proc. International Symposium on Low Power Electronics and Design ISLPED '99*. San Diego, 1999, 206–211.
10. Beneš, M., Nowick, S. M. and Wolfe, A. A fast asynchronous decompression circuit for embedded processors. In *Proc. Fourth International Symposium on Advanced Research in Asynchronous Circuits and Systems ASYNC98*. San Diego, 1998, 43–56.
11. Breternitz, M. Jr. and Smith, R. Enhanced compression techniques to simplify program decompression and execution. In *Proc. International Conference on Computer Design ICCD'97*. Austin, 1997, 170–176.
12. Laketsas, H., Henkel, J. and Wolf, W. Code compression for low power embedded system design. In *Proc. Design Automation Conference ACM/IEEE*. San Diego, 2000, 430–439.
13. Lefturgy, C. R., Piccininni, E. and Mudge, T. N. Reducing code size with run-time decompression. In *Proc. Sixth International Symposium on High-Performance Computer Architecture*. Toulouse, 2000, 218–227.
14. *Codepack: PowerPC Code Compression Utility User's Manual, Version 3.0*. IBM, 1998.
15. ARM Instruction Set Quick Reference Card v2.1. Online documentation; http://www.arm.com/pdfs/QRC0001H_rvct_v2.1_arm.pdf
16. iPAQ H3000 Pocket PC Reference Guide. Online documentation; http://h200005. www2.hp.com/bc/docs/support/UCR/ SupportManual/ TPM_177711-001/ TPM_177711-001.pdf

Kahendkoodi pakkimine otsustuspuude abil

Tamás Gergely, Ferenc Havasi ja Tibor Gyimóthy

Failide pakkimisel kasutatakse reeglina pakkimismeetodit, mis sõltumata andmete tüübist (kahendandmed, tekst, kahendkood) pakib kõik andmed, enamasti sama efektiivsusega. Andmete konkreetse tüübi spetsiaalne pakkija tihendaks andmeesitust failis paremini. Mitme sellise pakkija rakendamine erinevat tüüpi andmeid esitavas failis peaks andma parema pakkimistulemuse. Toodud ideest lähtudes on seatud eesmärgiks kahendkoodi efektiivse pakkimismeetodi koostamine. Enamik pakkimismeetodeist jaotub kaheks osaks: mudeliks ja kodeerijaks. Artiklis on esitatud otsustuspuudel põhinev modelleerimisviis. On katsetatud uut mudelit aritmeetilise kodeerijaga paaris failisüsteemile JFFS2 ARM-protsessoriga PDA-seadmete Linuxi distributsioonis. Algupärane failisüsteem kasutab failide pakkimiseks zlib-meetodit, uus süsteem erinevaid tihendamismeetodeid. Tulemused on paljulubavad: sõltuvalt meetodi häälestamisest säästetakse välismälu vähemalt 12,6% rohkem kui sama 13 MB suurust faili zlib-meetodiga pakkides. Failide paigaldamise kiirus langeb seejuures ainult kuni kolm korda.